



महाराष्ट्र राज्य तंत्रशिक्षण मंडळ, मुंबई
(स्वायत्त्व) (ISO 9001:2015) (ISO/IEC 27001:2013)

अभियांत्रिकी आणि तंत्रज्ञान पदविका

शिक्षण पुस्तिका

(Learning Material)

DATA STRUCTURE USING C

313301

K SCHEME

डेटा स्ट्रक्चर युसिंग सी (c)

संगणक अभियांत्रिकी
(के स्कीम)

मराठी-इंग्रजी (द्विभाषिक) माध्यम
(अभियांत्रिकी व तंत्रज्ञानातील तिसरे सत्र पदविका)
MagicStudio

शिक्षण पुस्तिका

(Learning Material)

डेटा स्ट्रक्चर युजींग C

Data Structure Using C

(313301)

संगणक अभियांत्रिकी गट

(के-स्कीम)

K-Scheme

मराठी-इंग्रजी (द्विभाषिक) माध्यम

(अभियांत्रिकी व तंत्रज्ञानातील तिसरे सत्र पदविका)



महाराष्ट्र राज्य तंत्रशिक्षण मंडळ, मुंबई

(स्वायत्त)(ISO 9001:2015) (ISO/IEC 27001:2013)

डेटा स्ट्रक्चर युजींग C
Data Structure Using C
(313301)

मार्गदर्शक

प्रा. विजय नामदेवराव कुकरे
विभागप्रमुख, संगणक अभियांत्रिकी

प्रकल्प समन्वयक

प्रा. परवेझ वाघेला
विभागप्रमुख, संगणक अभियांत्रिकी

संकलक

प्रा. परवेझ वाघेला
विभागप्रमुख, संगणक अभियांत्रिकी
आनंद आवारे
अधिव्याख्याता संगणक विभाग
अम्रिता राठोड
अधिव्याख्याता संगणक विभाग
शीतल जाधव
अधिव्याख्याता संगणक विभाग
प्रतीक गुरव
अधिव्याख्याता संगणक विभाग



महाराष्ट्र राज्य तंत्र शिक्षण मंडळ

(स्वायत्त) (ISO: ९००१:२०१५) (ISO/IES: २७००१-२०१३)

शासकीय तंत्रनिकेतन इमारत, चौथा मजला, ४९, खेरवाडी, बांद्रा (पूर्व), मुंबई - ४०० ०५१.

दूरध्वनी क्र.: ०२२-६२५४२१७० / १६१

Email : director@msbte.com

Web : www.msbte.org.in



प्रास्ताविक

महाराष्ट्र राज्यातील पदविका स्तरावरील तंत्रशिक्षणामध्ये विद्यार्थ्यांचे रोजगार कौशल्य विकसित करून विद्यार्थ्यांचा सर्वांगीण विकास घडवून आणण्याकरिता महाराष्ट्र राज्य तंत्रशिक्षण मंडळ कटिबद्ध आहे. उद्योगधंद्यातील बदलत्या तंत्रज्ञानाशी संबंधित गरजा लक्षात घेऊन महाराष्ट्र राज्य तंत्र शिक्षण मंडळाकडून पदविका अभ्यासक्रम वेळोवेळी अद्यावत करण्यात येतो. अभियांत्रिकी पदविका अभ्यासक्रम शिकत असतांना संकल्पनात्मक ज्ञान, सुसंगत संदर्भ, प्रश्न विचारणे, विश्वसनिय पुरावे, कारणमीमांसा आणि सुस्पष्ट निकष यांचा वापर करून अर्थाची उकल करण्याची, विश्लेषण व मूल्यमापन करण्याची तसेच तर्काने अनुमान काढण्याची क्षमता म्हणजेच चिकित्सक विचार विद्यार्थ्यांमध्ये अधिक दृढ होतील असा मला विश्वास आहे. जेव्हा विद्यार्थी ज्ञान मिळवण्याच्या माध्यमाशी पूर्णपणे परिचित आणि सोयीस्कर असतात, तेव्हा त्यांच्यासाठी वर्गातील चर्चेत भाग घेणे सोपे होते, संकल्पनात्मक व सैद्धांतिक बाबींचे आकलन परिपूर्ण होते, संज्ञानात्मक क्षमता सुधारते आणि त्यांचा आत्मविश्वास देखील वाढतो या सर्व गोष्टींचा विचार करून मंडळाकडून शैक्षणिक सामुग्रीची निर्मिती करण्यात आलेली आहे. भारत देश हा खेड्यापाड्यातून विकसित झालेला देश असून ग्रामीण भागातील विद्यार्थ्यांना तांत्रिक शिक्षण घेतांना भाषेचा अडसर न येता तांत्रिक बाबींचा आशय समजून घेणे शक्य होईल या दृष्टिकोनातून महाराष्ट्र राज्य तंत्र शिक्षण मंडळाने पदविका स्तरावरील तांत्रिक शिक्षणाकरिता विद्यार्थ्यांना मराठी-इंग्रजी द्विभाषिक माध्यमाचा पर्याय शैक्षणिक वर्ष २०२१-२२ पासून उपलब्ध करून दिलेला आहे.

राष्ट्रीय शैक्षणिक धोरण-२०२० प्रादेशिक भाषेतील शिक्षणास प्रोत्साहन देते, ज्यामुळे विद्यार्थ्यांना तांत्रिक अभ्यासक्रमासाठी प्रादेशिक भाषांतुन शिक्षणाचे माध्यम निवडता येते. सदर धोरणामुळे प्रादेशिक भाषांमध्ये तांत्रिक सामग्री आणि अभ्यास सामग्रीचा विकास आणि भाषांतर निर्माण करण्याची आवश्यकता आहे. त्यास अनुसरून मंडळाने मराठी-इंग्रजी द्विभाषिक माध्यमाचा पर्याय द्वितीय व तृतीय वर्षाकरिताही उपलब्ध करून देण्यात आला आहे. तसेच त्याकरिताची शैक्षणिक सामग्रीही संबंधीत भागधारकरांना उपलब्ध करून देण्यात येत आहे.

पदविका स्तरावरील तंत्रशिक्षण अधिक दर्जेदार करण्यासाठी महाराष्ट्रातील अनुभवी व तज्ञ अध्यापकांनी व्यवहारिक मराठी भाषा व इंग्रजी भाषेतील तांत्रिक शब्दावली यांचा वापर करून मराठी - इंग्रजी भाषेचा सुवर्णमध्य साधण्याचा प्रयत्न केलेला आहे. मंडळाच्या स्तरावर गठीत सुकाणू समितीमार्फत सदर शैक्षणिक सामुग्रीचा दर्जा, तसेच इतर बाबींची तपासणी करण्यात आलेली आहे. त्यामुळे सदर शैक्षणिक सामुग्री अधिक सम्पन्न झालेली असून विद्यार्थी त्यांच्या व्यक्तिमत्त्वाचा सुसंवादी आणि सर्वांगीण विकास साधतील. परिणामतः विश्वस्तरीय मनुष्यबळाच्या गरजा पूर्ण करण्यात महाराष्ट्र राज्य अग्रेसर राहील व पर्यायाने राष्ट्रनिर्मिती करीता निश्चितच हातभार लागेल, असा मला विश्वास आहे.

अभियांत्रिकी पदविका अभ्यासक्रमातील प्रमुख विषयांची मराठी-इंग्रजी द्विभाषिक शैक्षणिक सामुग्री बनविण्यासाठी अध्यापक व सुकाणू समितीचे सदस्य यांनी दर्शविलेले समर्पण व वचनबद्धता कौतुकास पात्र आहे, या सर्वांचे मी मनः पूवक अभिनंदन करतो !



(प्रमोद नाईक)

संचालक

म. रा. तंत्र शिक्षण मंडळ, मुंबई.

अनुक्रमणिका

अ. नु.	युनिटचे नाव	पृष्ठ क्रमांक
1	डेटा स्ट्रक्चरचा परिचय (Introduction to Data Structure)	1
2	सर्चिंग आणि सॉर्टिंग (Searching and Sorting)	6
3	लिंकड लिस्ट (Linked List)	27
4	स्टॅक (Stack)	37
5	क्यू (Queue)	68
6	ट्री (Tree)	84

युनिट -1

डेटा स्ट्रक्चरचा परिचय

(Introduction to Data Structure)

विषय निष्पत्ती (Course Outcome): ऑरेवर मूलभूत ऑपरेशन्स करा.

घटक निष्पत्ती (Theory Learning Outcomes):

1. दिलेल्या प्रकारच्या डेटा स्ट्रक्चर्सचे त्यांची वैशिष्ट्ये आणि जागेच्या आधारावर वर्गीकरण करा.
2. दिलेल्या प्रकारच्या डेटा स्ट्रक्चरवर ऑपरेशन्स करा.

1.1 डेटा चा परिचय:(Introduction to Data Structure)

डेटा(Data): डेटा ही मूलभूत entity किंवा तथ्य आहे जी गणना किंवा हेरफेर (manipulation) प्रक्रियेत वापरली जाते. डेटा मूल्य किंवा मूल्यांचा संच आहे जो कोणताही अर्थ देत नाही. ही सर्व साधारण पणे कच्ची वस्तुस्थिती आहे.

उदाहरणार्थ

1.1.1 डेटा टाइप (Data type) : डेटा प्रकार हा एक शब्द आहे जो गणनेत दिसू शकणार्या डेटाच्या प्रकारास संदर्भित करतो. प्रत्येक प्रोग्रामिंग भाषेचा स्वतःचा अंतर्निहित डेटा प्रकारांचा संच असतो.

उदाहरण: डेटा

Table: 1.1

डेटा	डेटा टाइप (Data type)
34	संख्यात्मक
21,1,3	पूर्णांकांची सरणी
12/05/2000	तारीख (Date)

सी "C" मध्ये, मूलभूत डेटा प्रकार खालीलप्रमाणे आहेत: इंट(int), लॉंग(long), कॅरेक्टर(character), फ्लोट(float), डबल(double), व्हॉइड(void) इ.

1.1.2 डेटा रचनेची व्याख्या (Definition of Data Structure):

"डेटा स्ट्रक्चर हे डेटाच्या विशिष्ट संघटनेचे तार्किक किंवा गणितीय मॉडेल आहे".

डेटा स्ट्रक्चर = संघटित डेटा + अनुमत ऑपरेशन्स

1.1.3 डेटा स्ट्रक्चरची गरज(Need of Data Structure)

- एका डेटा घटकांचा दुसर् या डेटा घटकांशी असलेला संबंध समजून घेणे आणि मेमरीमध्ये त्याचे आयोजन करणे.
- डेटा स्ट्रक्चर डेटाचे विश्लेषण करण्यास, संग्रहित करण्यास आणि तार्किक किंवा गणितीय पद्धतीने व्यवस्थित करण्यास मदत करते.
- डेटा संरचना आम्हाला एक महत्त्वपूर्ण लक्ष्य साध्य करण्यास अनुमती देते: घटक पुनर्वापर.

1.1.4 ऍबस्ट्रॅक्ट डेटाचा प्रकार(Abtract data type)

- ऍबस्ट्रॅक्ट डेटा प्रकार (एडीटी) ऑब्जेक्ट्ससाठी एक प्रकार (किंवा वर्ग) आहे ज्यांचे वर्तन मूल्यांच्या संच (Set) आणि ऑपरेशन्सच्या संचाद्वारे परिभाषित केले जाते.
- एडीटीच्या व्याख्येत फक्त कोणती कारवाई करायची याचा उल्लेख आहे पण या ऑपरेशन्सची इम्प्लिमेंटेशन (implementation) कशी केली जाईल याचा उल्लेख नाही.
- मेमरीमध्ये डेटा कसा आयोजित(Organize) केला जाईल आणि ऑपरेशन्स च्या इम्प्लिमेंटेशनसाठी कोणते अल्गोरिदम वापरले जातील हे यात नमूद केलेले नाही.

- त्याला "ऍबस्ट्रॅक्ट" (ABSTRACT) म्हणतात कारण ते इम्प्लिमेंटेशन-स्वतंत्र दृष्टीकोन देते.
- केवळ ESSENTIALS पुरविणे आणि तपशील लपविणे(HIDING DETAILS) या प्रक्रियेला ऍबस्ट्रॅक्ट (ABSTRACT) म्हणतात.

1.1.5 एडीटीसी (Abstract data type) वैशिष्ट्ये(Features)

- **एनकॅप्सुलेशन(Encapsulation):** एडीटी डेटा आणि डेटावरील ऑपरेशन्स एकत्र समाविष्ट करते, जिथे डेटा बाहेरील जगाला उपलब्ध नसतो आणि त्यावर केवळ ऑपरेशन्सची परवानगी असते.
- **माहिती लपवणे(Hiding Details):** एडीटी वापरकर्त्यापासून अंतर्गत यंत्रणा लपवून ठेवत केवळ आवश्यक तपशील उघड करतात.

1.2 डेटा स्ट्रक्चरचे प्रकार (Types data structure)

डेटा आयटम कसे ऑपरेट केले जातात यावर आधारित, हे दोन व्यापक श्रेणींमध्ये वर्गीकृत केले जाईल.

1. लिनिअर डेटा स्ट्रक्चर (Linear data structure)
2. नॉन लिनिअर डेटा स्ट्रक्चर (Non linear data structure)

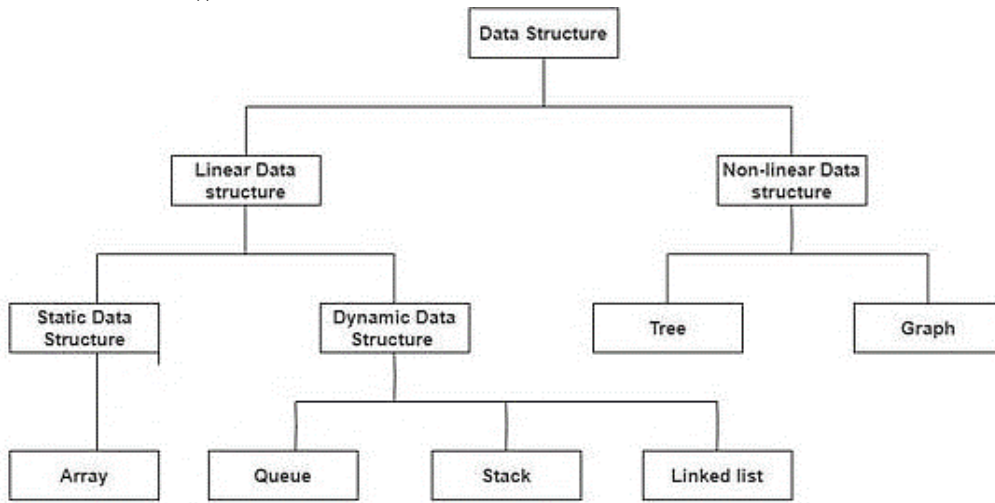


Fig.1.1: डेटा संरचनेचे वर्गीकरण (Classification of data structure)

1.2.1 लिनिअर डेटा स्ट्रक्चर(Linear data structure)

लिनिअर डेटा (Linear data) संरचना अनुक्रमिक पद्धतीने डेटा संग्रहित करतात, जिथे प्रत्येक घटक त्याच्या मागील आणि पुढील घटकाशी जोडला जातो.

उदाहरणे म्हणजे अरे(Array), स्टॅक(Stack), क्यू(Queue) आणि लिंकलिस्ट (Linked List) केलेली यादी.

a. **अरे (Array):** अरेची व्याख्या: सलग (Contiguous) मेमरी लोकेशन्सवर संग्रहित केलेल्या समान प्रकारच्या डेटा आयटमचा संग्रह म्हणून केली जाते.

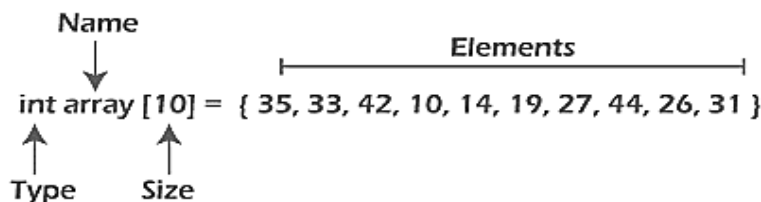


Fig. 1.2: अरे (Array)

b. **लिंकड लिस्ट (Linked list):** हे डेटा घटकांचे लिनिअर (linear) संकलन आहे. लिंक केलेल्या सूचीतील प्रत्येक घटकाला 'नोड' असे म्हणतात. प्रत्येक नोडमध्ये दोन फील्ड असतात. प्रथम माहिती (Information) माहिती संग्रहित करते दुसरे म्हणजे LINK/NEXT जे यादीतील पुढील नोडच्या ऍड्रेसशी जोडलेले आहे .

जोडलेल्या याद्या सूचीतील कोणत्याही स्थानावरून घटक (element) समाविष्ट (add) करण्यास किंवा काढून टाकण्यास (delete /remove) परवानगी देतात.

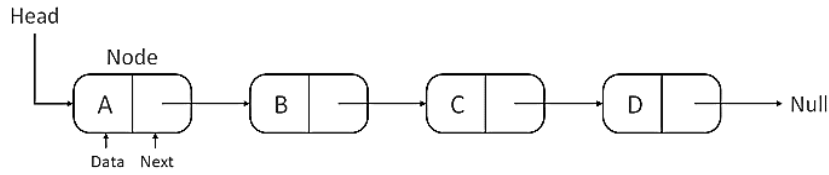


Fig. 1.3: लिंकड लिस्ट (Linked list)

c. **स्टॅक (Stack)** : एक लिनिअर डेटा संरचना आहे जी विशिष्ट क्रमाचे अनुसरण करते ज्यामध्ये ऑपरेशन केले जातात. ऑर्डर लिफो-(LIFO-लास्ट इन फर्स्ट आऊट) किंवा फिलो-(FILO-फर्स्ट इन लास्ट आऊट) असू शकते. लिफो चा अर्थ असा आहे की जो घटक शेवटी घातला जातो, तो प्रथम बाहेर येतो आणि फिलो चा अर्थ असा आहे की जो घटक प्रथम घातला जातो तो शेवटी येतो.

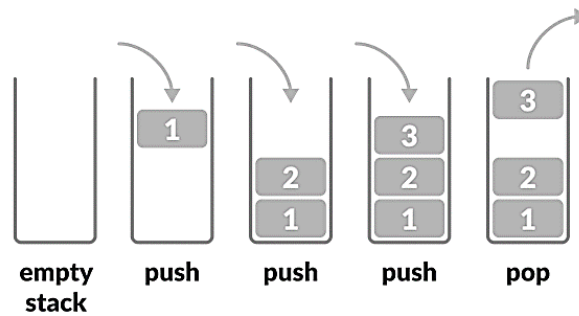


Fig. 1.4: स्टॅक (stack)

d. **क्यू (Queue)**: क्यू ही एक लिनिअर डेटा रचना आहे संगणक विज्ञानातील एक मूलभूत संकल्पना आहे जी विशिष्ट क्रमाने डेटा संग्रहित आणि व्यवस्थापित करण्यासाठी वापरली जाते. हे "फर्स्ट इन, फर्स्ट आऊट" (फिफो-FIFO) या तत्वाचे अनुसरण करते, जेथे रांगेत जोडलेला पहिला घटक काढून टाकला जातो.

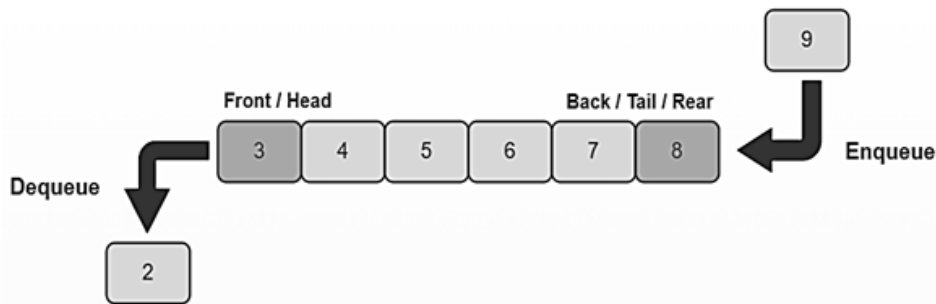


Fig. 1.5: क्यू (Queue)

1.2.2 नॉन लिनिअर डेटा स्ट्रक्चर (Non linear data structure):

डेटा स्ट्रक्चर जिथे डेटा घटक क्रमवार किंवा लिनिअर पद्धतीने व्यवस्थित केले जात नाहीत त्यांना नॉन-लिनिअर डेटा स्ट्रक्चर म्हणतात. नॉन-लिनिअर डेटा स्ट्रक्चरमध्ये, सिंगल लेव्हल चा समावेश नसतो. त्यामुळे केवळ एकाच धावात आपण सर्व घटकांना पार करू शकत नाही. लिनिअर डेटा स्ट्रक्चर च्या तुलनेत नॉन-लिनिअर डेटा स्ट्रक्चर अंमलात आणणे सोपे नाही. हे लिनिअर डेटा रचनेच्या तुलनेत संगणक मेमरी कार्यक्षमतेने वापरते. ट्री(Trees) आणि आलेख(Graph) ही त्याची उदाहरणे आहेत.

a. **ट्री (Trees)** : ट्री ही एक नॉन-लिनिअर डेटा स्ट्रक्चर आहे ज्यामध्ये विविध नोड्स एकत्र जोडलेले असतात. ट्रीची रचना श्रेणीबद्ध असते जी पालक (parent) आणि मुलासारखे (child) नाते तयार करते.

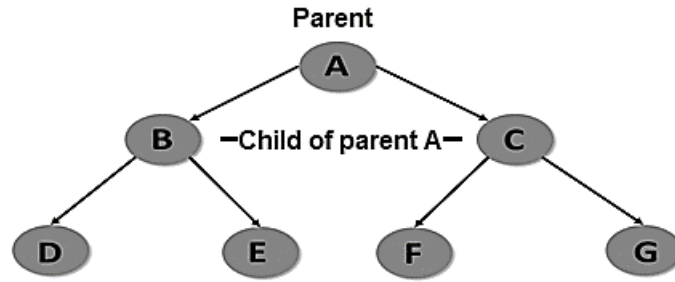


Fig. 1.6: ट्री(Trees)

- b. **आलेख (Graph):** आलेख हे नॉन-लिनियर प्रकारचे डेटा स्ट्रक्चर आहेत ज्यात विशिष्ट प्रमाणात **एज (Edge)** आणि **नोड (Node)** असतात. नोड्स मध्ये डेटा साठविण्यात येतो आणि **कडा(Edge)** संबंध दर्शवितात.

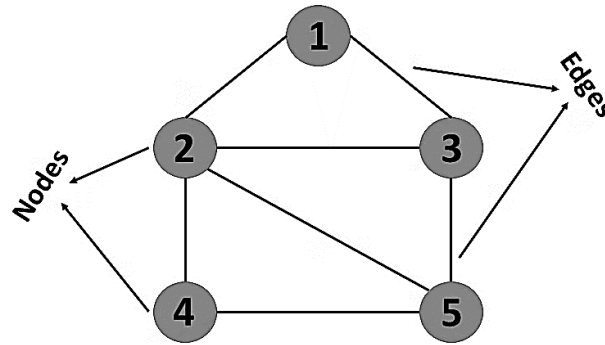


Fig. 1.7: ट्री (Trees)

1.3 डेटा स्ट्रक्चर वर ऑपरेशन्स (Operations on Data structure):

प्रत्येक डेटा स्ट्रक्चरमध्ये डेटाच्या हाताळणीसाठी वेगवेगळ्या प्रकारचे ऑपरेशन केले जाऊ शकतात. काही ऑपरेशन्स खाली दिले आहेत:

1. **ट्रॅवर्सिंग(Traversing):** डेटा स्ट्रक्चर पार करणे म्हणजे त्यामध्ये साठवलेल्या घटकाला भेट देणे. हे पद्धतशीरपणे डेटाला भेट देते. हे कोणत्याही प्रकारच्या डेटा स्ट्रक्चरसह केले जाऊ शकते.
2. **इन्सर्शन(Insertion):** ही अशी क्रिया आहे जी आम्ही सर्व डेटा-स्ट्रक्चरवर लागू केले जाऊ शकते. इन्सर्शन म्हणजे दिलेल्या डेटा स्ट्रक्चरमध्ये एक घटक जोडणे. आवश्यक डेटा-रचनेत आवश्यक घटक जोडला गेल्यास प्रवेशाचे (**Insertion**) ऑपरेशन यशस्वी होते. जेव्हा डेटा संरचनेचा आकार(size) पूर्ण (Full) असतो आणि जेव्हा डेटा-संरचनेत कोणताही अतिरिक्त घटक जोडण्यासाठी जागा नसते तेव्हा काही प्रकरणांमध्ये ते अयशस्वी ठरते.
3. **डिलीट(Delete):** ही अशी क्रिया आहे जी आम्ही सर्व डेटा-स्ट्रक्चरवर लागू करतो. डिलीट करणे म्हणजे दिलेल्या डेटा स्ट्रक्चरमधील एखादा घटक डिलीट करणे. डेटा रचनेतून आवश्यक घटक काढून टाकल्यास हटविण्याचे ऑपरेशन यशस्वी होते.
4. **शोध (Search):** शोध म्हणजे दिलेल्या डेटा-रचनेतील विशिष्ट घटक शोधणे. जेव्हा आवश्यक घटक सापडतो तेव्हा ते यशस्वी मानले जात

संदर्भ (References):

1. Data Structure Using C : Dr. E. Balagurusamy
2. Data structure and Algorithms : K. Mehlhorn
3. Data structure and Algorithms: Dr. Mizanur Rahman

4. Data Structure with c :Lipschutz Schaum Series.
5. <https://www.geeksforgeeks.org/introduction-to-data-structures/>
6. <https://www.javatpoint.com/data-structure-introduction>
7. https://www.w3schools.com/dsa/dsa_intro.php

युनिट 2

सर्चींग आणि सॉर्टींग

(Searching and Sorting)

विषय निष्पत्ती (Course Outcome): वेगवेगळ्या सर्चींग आणि सॉर्टींग पद्धती लागू करा.

घटक निष्पत्ती (Theory Learning Outcomes):

1. वेगवेगळ्या सर्च तंत्रांचा वापर करून दिलेली की शोधण्यासाठी अल्गोरिदम विकसित करा.
2. दिलेल्या पद्धतीचा वापर करून डेटा क्रमवारी लावण्यासाठी अल्गोरिदम तयार करा

2.1 डेटा स्ट्रक्चरवरील ऑपरेशन्स:(Operations on Data structure)

आम्ही युनिट 1 मध्ये चर्चा केल्याप्रमाणे , डेटा स्ट्रक्चरवर विविध ऑपरेशन्स केल्या जाऊ शकतात जसे की-

1. **इन्सर्शन(insertion)** : डेटा स्ट्रक्चरमध्ये घटक जोडणे.
2. **डिलीट(delete)**: डेटा स्ट्रक्चरमधून घटक काढून टाकणे.
3. **ट्रॅव्हर्सिंग (traversing)**: डेटा स्ट्रक्चरमधून प्रत्येक घटकामध्ये प्रवेश करणे किंवा भेट देणे.
4. **सर्च(search)**: दिलेल्या डेटा स्ट्रक्चरमध्ये घटक उपस्थित आहे की नाही हे तपासणी करणे.
5. **वर्गीकरण(sorting)** : काही तार्किक क्रमाने डेटा स्ट्रक्चरचे घटक व्यवस्थित करणे.

2.2 सर्चिंग (Searching):

सर्चिंग ही डेटाच्या संग्रहामध्ये विशिष्ट घटक किंवा आयटम सर्चण्याची मूलभूत प्रक्रिया आहे.

- डेटाचा हा संग्रह विविध फॉर्म घेऊ शकतो, जसे की अरे(Array), लिस्ट(List), ट्रीज(Trees) किंवा इतर स्ट्रक्चर प्रतिनिधित्व .
- शोधाचा प्राथमिक उद्देश डेटामध्ये इच्छित घटक अस्तित्वात आहे की नाही हे निर्धारित करणे आणि तसे असल्यास, त्याचे अचूक स्थान ओळखणे किंवा ते पुनर्प्राप्त करणे हे आहे.
- माहिती पुनर्प्राप्ती, डेटा विश्लेषण, निर्णय घेण्याची प्रक्रिया आणि बरेच काही यासह विविध संगणकीय कार्ये आणि वास्तविक-जगातील अनुप्रयोगांमध्ये ते महत्त्वपूर्ण भूमिका बजावते.

2.2.1 सर्चिंग अल्गोरिदम (Searching Algorithm) :

सर्चिंग करण्याचे अल्गोरिदम हे घटक तपासण्यासाठी किंवा ते संचयित केलेल्या कोणत्याही डेटा स्ट्रक्चरमधून घटक पुनर्प्राप्त(Access) करण्यासाठी डिझाइन केलेले आहेत.

सर्चिंग ही यादीतील काही विशिष्ट घटक सर्चण्याची प्रक्रिया आहे. जर सूचीमध्ये घटक उपस्थित असेल, तर प्रक्रियेस यशस्वी म्हटले जाते आणि प्रक्रिया त्या घटकाचे स्थान परत करते. अन्यथा, सर्चअयशस्वी म्हटले जातेखाली काही सर्चअल्गोरिदम आहेत:

1. लिनियर सर्च (Linear Search)
2. बायनरी सर्च (Binary Search)

1. **लिनियर सर्च(Linear Search)**: लिनियर सर्च, ज्याला अनुक्रमिक सर्चदेखील म्हणतात, हा सर्वात सोपा आणि सर्वात सरळ सर्चअल्गोरिदम आहे. लिनियर सर्चमध्ये, आम्ही फक्त सूची पूर्णपणे पार करतो आणि सूचीतील प्रत्येक घटक ज्याचे स्थान शोधायचे आहे त्या आयटमशी जुळणी आढळल्यास, आयटमचे स्थान परत केले जाते; अन्यथा, अल्गोरिदम NULL परत करेल. अक्रमित सूचीमधून घटक सर्चिंग करण्यासाठी याचा मोठ्या प्रमाणावर वापर केला जातो, म्हणजे ज्या सूचीमध्ये आयटमची सॉर्ट (Sorted List) लावलेली नाही. लिनियर सर्चची सर्वात वस्त-केस (Worst case) वेळ जटिलता(Time Complexity) $O(n)$ आहे.

अरे $arr[] = \{10, 50, 30, 70, 80, 20, 90, 40\}$ आणि की = 30 विचारात घ्या

लिनियर सर्चच्या इम्प्लिमेंटेशनसाठी वापरल्या जाणाऱ्या स्टेप्स खालीलप्रमाणे दिलेल्या आहेत -

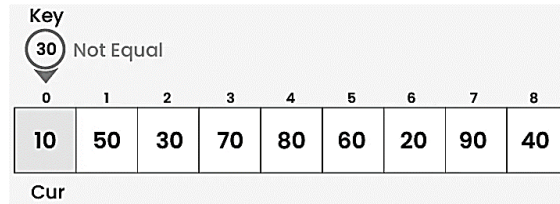
1. प्रथम, आपण फॉर लूप (For Loop) वापरून अरे एलिमेंट्स ट्रॅव्हर्स (Traverse) करावे लागतील.
2. फॉर लूपच्या प्रत्येक रिकर्षणमध्ये, सर्चघटकाची वर्तमान अरे घटकाशी तुलना करा आणि -
 - a. घटक जुळत असल्यास, संबंधित अरे घटकाची अनुक्रमणिका परत करा.
 - b. जर घटक जुळत नसेल, तर पुढील घटकाकडे जा.

3. दिलेल्या अर्रेमध्ये कोणतीही जुळणी नसल्यास किंवा सर्चघटक उपस्थित नसल्यास -1 परत करा .
4. अल्गोरिदम संग्रहातील प्रत्येक घटकाचे एक एक करून परीक्षण करते, प्रत्येक घटकाला तुम्ही सर्चत असलेल्या की साठी संभाव्य जुळणी मानून.

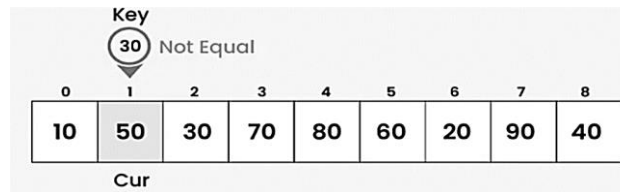
तुम्ही सर्च करत असलेल्या की सारखाच घटक आढळल्यास, सर्चयशस्वी होतो आणि तो कीची अनुक्रमणिका परत करतो. जर ते सर्व घटकांमधून गेले आणि त्यापैकी एकहीशी जुळत नसेल, तर याचा अर्थ "कोणताही जुळणी सापडली नाही". अनुक्रमणिका 0 सह पहिल्या घटकापासून प्रारंभ करा आणि अर्रे अर्रे[]च्या प्रत्येक घटकाशी की घटकाची तुलना करा.

प्रथम घटक arr[0] शी मुख्य घटकाची तुलना करा, कारण समान नाही, इटरेटर पुढील घटकाकडे जातो.

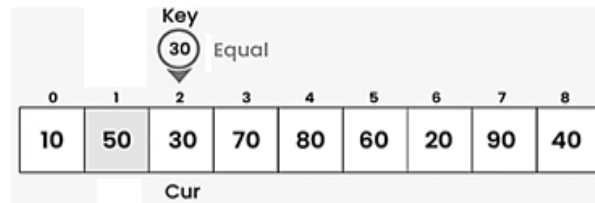
Eg:



1. पुढील घटक arr[1] शी की तुलना करणे. समान नसल्यामुळे, इटरेटर पुढील घटकाकडे जातो.



2. आता arr[2] ची कीशी तुलना करताना, मूल्य जुळते. त्यामुळे लिनियर सर्चा अल्गोरिदम एक यशस्वी संदेश देईल आणि जेव्हा की सापडेल तेव्हा घटकाची अनुक्रमणिका परत करेल.



लिनियर सर्च कधी वापरायचा:

- जेव्हा डेटाचा लहान संग्रह असतो.
- जेव्हा डेटा क्रमबद्ध नसतो.

Program for Linear Search

```
#include <stdio.h>
#include <conio.h>
int linearSearch(int arr[], int size, int target)
{
    // Traverse through the array
    int i;
    for (i = 0; i < size; i++)
    {
        // If the element is found, return the index
        if (arr[i] == target) {
            return i; // Element found at index i
        }
    }
}
```

```

    }
    return -1; // Element not found
}
int main() {
    int n,i, target, result,arr[100];
    // Input the number of elements in the array
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    // Input the elements of the array
    printf("Enter %d elements: \n", n);
    for ( i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    // Input the element to be searched
    printf("Enter the element to search: ");
    scanf("%d", &target);
    // Call the linear search function
    result = linearSearch(arr, n, target);
    // Output the result
    if (result == -1) {
        printf("Element %d not found in the array.\n", target);
    } else {
        printf("Element %d found at index %d.\n", target, result);
    }
    getch();
    return 0;
}

```

Output:

```

Enter the number of elements: 5
Enter 5 elements: 3 4 2 8 5
Enter the element to search:8
Element 8 found at index 3

```

2. बायनरी सर्च(Binary search):

बायनरी सर्च हे सर्चमध्यांतर अर्ध्यामध्ये वारंवार विभाजित करून सॉर्ट केलेल्या अॅरेमध्ये वापरलेले सर्चअल्गोरिदम म्हणून परिभाषित केले आहे. बायनरी सर्चची कल्पना म्हणजे अॅरेची सॉर्ट लावलेली माहिती वापरणे आणि वेळेची जटिलता $O(\log N)$ पर्यंत कमी करणे.

बायनरी सर्च हे सर्चतंत्र आहे जे सॉर्ट केलेल्या सूचींवर कार्यक्षमतेने कार्य करते. म्हणून, बायनरी सर्चतंत्राचा वापर करून काही सूचीमध्ये घटक सर्चण्यासाठी, आम्ही सूची सॉर्टत आहे याची खात्री केली पाहिजे.

बायनरी सर्चविभागणी आणि जिंकण्याच्या दृष्टिकोनाचे अनुसरण करते ज्यामध्ये सूची दोन भागांमध्ये विभागली जाते आणि आयटमची सूचीच्या मधल्या घटकाशी तुलना केली जाते. जर जुळणी आढळली तर, मधल्या घटकाचे स्थान परत केले जाईल. अन्यथा, आम्ही सामान्याद्वारे उत्पादित केलेल्या निकालावर अवलंबून कोणत्याही अर्ध्या भागांमध्ये सर्चतो.

अल्गोरिदम (Algorithm):

Binary_Search(a, Lower_bound, upper_bound, val) // 'a' हा दिलेला अरे आहे, 'lower_bound' हा पहिल्या अरे घटकाचा इंडेक्स आहे, 'upper_bound' हा शेवटच्या अरे घटकाचा इंडेक्स आहे, 'val' हे मूल्य आहे सर्चणे

Step 1: set beg = lower_bound, end = upper_bound, pos = - 1

Step 2: repeat steps 3 and 4 while beg <=end

Step 3: set mid = (beg + end)/2

Step 4: if a[mid] = val

set pos = mid

print pos

go to step 6

else if a[mid] > val

set end = mid - 1

else

set beg = mid + 1

[end of if]

[end of loop]

Step 5: if pos = -1

print "value is not present in the array"

[end of if]

Step 6: exit

बायनरी सर्चअल्गोरिदमचे कार्य समजून घेण्यासाठी, एक क्रमबद्ध अरे घेऊ. उदाहरणासह बायनरी शोधाचे कार्य समजून घेणे सोपे होईल .

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

सर्चण्यासाठी घटक द्या, **K = 56**

अरेच्या **मध्याची** गणना करण्यासाठी आपल्याला खालील सूत्र वापरावे लागेल -

1. **mid** = (beg + end)/2

तर, दिलेल्या अरेमध्ये -

beg = 0

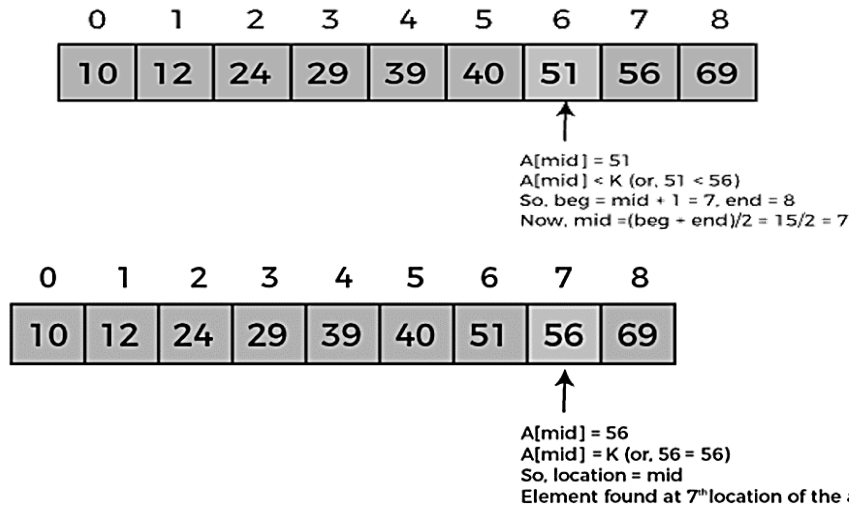
end = 8

mid = (0 + 8)/2 = 4. तर, 4 हा अरेचा मध्य आहे.

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69



A[mid] = 39
A[mid] < K (or, 39 < 56)
So, beg = mid + 1 = 5, end = 8
Now, mid = (beg + end)/2 = 13/2 = 6



आता, सर्चण्यासाठी घटक सापडला आहे. त्यामुळे अल्गोरिदम जुळलेल्या घटकाची अनुक्रमणिका परत करेल. बायनरी सर्चची जटिलता

आता, सर्वोत्तम केस, सरासरी केस आणि सर्वात वाईट केसमध्ये बायनरी शोधाची वेळ जटिलता पाहू. आपण बायनरी शोधाची जागा जटिलता देखील पाहू.

a. टाइम कॉम्प्लेक्सिटी (Time Complexity)

Table 2.1: Time complexity

केस	वेळेची गुंतागुंत
सर्वोत्तम केस (बेस्ट केस)	$O(1)$
सरासरी केस (अवरेज केस)	$O(\log n)$
सर्वात वाईट केस (वष्ट केस)	$O(\log n)$

- **सर्वोत्कृष्ट टाइम कॉम्प्लेक्सिटी** - बायनरी सर्चमध्ये, जेव्हा शोधाचा घटक पहिल्या तुलनेमध्ये सापडतो, म्हणजे जेव्हा पहिला मधला घटक शोधाचा घटक असतो. तेव्हा सर्वोत्तम केस उद्भवते. बायनरी शोधाची सर्वोत्तम-केस टाइम कॉम्प्लेक्सिटी $O(1)$ आहे.
- **सरासरी टाइम कॉम्प्लेक्सिटी** - बायनरी शोधाची सरासरी टाइम कॉम्प्लेक्सिटी $O(\log n)$ आहे.
- **सर्वात वाईट टाइम कॉम्प्लेक्सिटी** - बायनरी सर्चमध्ये, सर्वात वाईट परिस्थिती उद्भवते, जेव्हा आपल्याला सर्चजागा कमी करत राहावे लागते जोपर्यंत फक्त एक घटक मिळत नाही. बायनरी शोधाची सर्वात वाईट-केस टाइम कॉम्प्लेक्सिटी $O(\log n)$ आहे.

b. स्पेस कॉम्प्लेक्सिटी (Space Complexity): बायनरी शोधाची स्पेस कॉम्प्लेक्सिटी $O(1)$ आहे.

Program:

```
#include <stdio.h>
```

```
#include <conio.h>
```

```
// Function to perform binary search on a sorted array
```

```
int binarySearch(int arr[], int size, int target) {
```

```

int low = 0, high = size - 1;
while (low <= high) {
    // Find the middle element
    int mid = low + (high - low) / 2;
    // Check if the target is at the mid
    if (arr[mid] == target) {
        return mid; // Element found at index mid
    }
    // If the target is greater than the mid, search the right half
    if (arr[mid] < target) {
        low = mid + 1;
    }
    // If the target is smaller than the mid, search the left half
    else {
        high = mid - 1;
    }
}
return -1; // Target not found
}

int main() {
    int n, target, result, arr[100], i;
    // Input the number of elements in the array
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    // Input the elements of the array (must be sorted)
    printf("Enter %d sorted elements: \n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    // Input the element to be searched
    printf("Enter the element to search: ");
    scanf("%d", &target);
    // Call the binary search function
    result = binarySearch(arr, n, target);
    // Output the result
    if (result == -1) {
        printf("Element %d not found in the array.\n", target);
    } else {
        printf("Element %d found at index %d.\n", target, result);
    }
    getch();
    return 0;
}

```

}

Output:

Enter the number of elements:6

Enter 6 sorted elements: 1 3 4 6 8 9

Enter the element to search:6

Element 6 found at index 3.

2.2.2 सॉर्टिंग (Sorting)

सॉर्ट (Sort) करणे ही अरेच्या घटकांची मांडणी करण्याची प्रक्रिया आहे जेणेकरून ते चढत्या किंवा उतरत्या क्रमाने ठेवता येतील.

अरेचा विचार करा;

int A[10] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9 }

चढत्या क्रमाने लावलेला अरे याप्रमाणे दिसेल:

अ [] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }

अशी अनेक तंत्रे आहेत ज्याचा वापर करून अरे सॉर्ट केला जाऊ शकतो:

1. बबल सॉर्ट (Bubble Sort)
2. इन्सर्शन सॉर्ट (Insertion Sort)
3. सिलेक्शन सॉर्ट (Selection Sort)
4. क्विक सॉर्ट (Quick Sort)
5. मर्ज सॉर्ट (Merge Sort)

1. बबल सॉर्ट (Bubble Sort)

बबल सॉर्ट तंत्रात, सूचीतील प्रत्येक घटकाची त्याच्या जवळील घटकाशी तुलना केली जाते. अशा प्रकारे जर सूची A मध्ये n घटक असतील, तर A[0] ची A[1] शी तुलना केली जाते, A[1] ची A[2] शी तुलना केली जाते आणि असेच. पहिला घटक दुसऱ्यापेक्षा मोठा असल्यास तुलना केल्यानंतर, दोन घटकांची अदलाबदल केली जाते.

बबल सॉर्ट तंत्र वापरून, वर्गीकरण पास किंवा रिकर्षणमध्ये केले जाते. अशा प्रकारे प्रत्येक रिकर्षणच्या शेवटी, सर्वात वजनदार घटक सूचीमध्ये त्याच्या योग्य ठिकाणी ठेवला जातो. दुसऱ्या शब्दांत, सूचीतील सर्वात मोठा घटक शेवटी येतो.

बबल सॉर्ट अल्गोरिदममध्ये:

1. डावीकडून मार्गक्रमण करा आणि समीप घटकांची तुलना करा आणि वरचा भाग उजव्या बाजूला ठेवला आहे.
2. अशा प्रकारे, सर्वात मोठा घटक प्रथम उजव्या टोकाला हलविला जातो.
3. ही प्रक्रिया नंतर दुसरी सर्वात मोठा घटक सर्च करण्यासाठी आणि तो योग्य जागेवर ठेवण्यासाठी आणि डेटाची सॉर्ट होई पर्यंत चालू ठेवली जाते.

अल्गोरिदम

Step 1: For i = 0 to N-1 repeat Step 2

Step 2: For J = i + 1 to N – I repeat

Step 3: if A[J] > A[i]

Swap A[J] and A[i]

[End of Inner for loop]

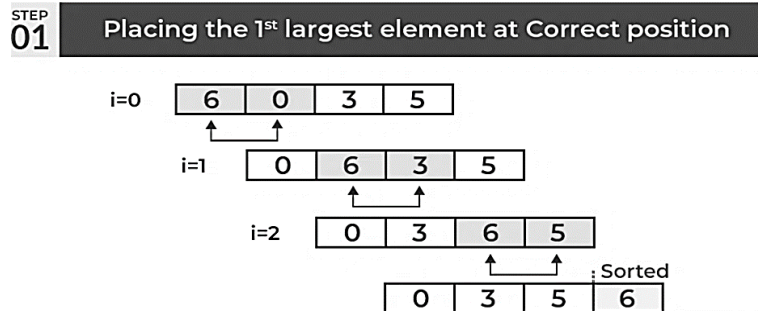
[End if Outer for loop]

Step 4: Exit

पहिल्या लूपमध्ये, आपण [0] व्या घटकापासून सुरुवात करतो आणि पुढील लूपमध्ये, आपण जवळचा घटकापासून सुरुवात करतो. आतील लूप बॉडीमध्ये, आम्ही प्रत्येक शेजारील घटकांची तुलना करतो आणि जर ते क्रमाने नसतील तर त्यांची अदलाबदल करतो. बाह्य लूपच्या प्रत्येक रिकर्षणच्या शेवटी, सर्वात मोठा घटक शेवटी येतो.

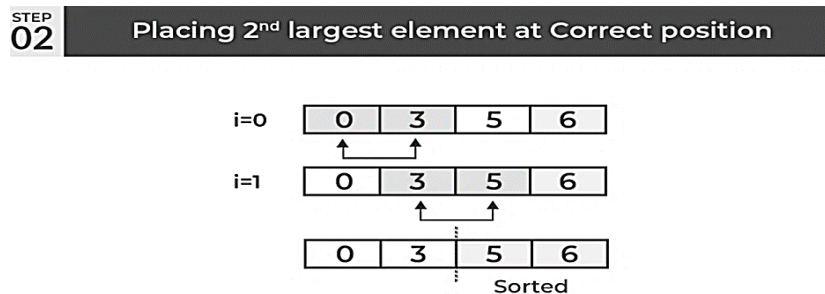
पहिला पास:

सर्वात मोठा घटक त्याच्या योग्य स्थितीत ठेवला आहे, म्हणजे, अंरच्या शेवटी.



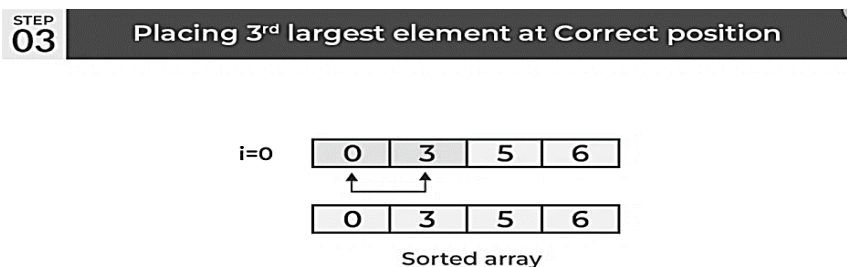
दुसरा पास:

दुसरा सर्वात मोठा घटक योग्य स्थानावर ठेवा



तिसरा पास:

उर्वरित दोन घटक त्यांच्या योग्य स्थानावर ठेवा.



एकूण पासेस: $n-1$

एकूण तुलना: $n*(n-1)/2$

बबल सॉर्टचे जटिलता विश्लेषण :

टाइम कॉम्प्लेक्सिटी (Time Complexity): $O(N^2)$

स्पेस कॉम्प्लेक्सिटी (Space Complexity): $O(1)$

Program:

#include <stdio.h>

```
#include<conio.h>
// Function to perform bubble sort
void bubbleSort(int arr[], int size) {
    // Loop through all elements in the array
    int i,j;
    for (i = 0; i < size - 1; i++) {
        // Last i elements are already sorted, so we skip them
        for (j = 0; j < size - i - 1; j++) {
            // Swap if the element found is greater than the next element
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
// Function to print the array
void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int n,i,arr[100];
    // Input the number of elements in the array
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    // Input the elements of the array
    printf("Enter %d elements: \n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    // Perform the bubble sort
    bubbleSort(arr, n);
    // Print the sorted array
    printf("Sorted array: \n");
    printArray(arr, n);
    getch();
}
```

```
return 0;
}
```

Output:

Enter the number of elements:5

Enter 5 elements: 5 3 0 7 1

Sorted array:

0 1 3 5 7

2. इन्सर्शन सॉर्ट (Insertion Sort)

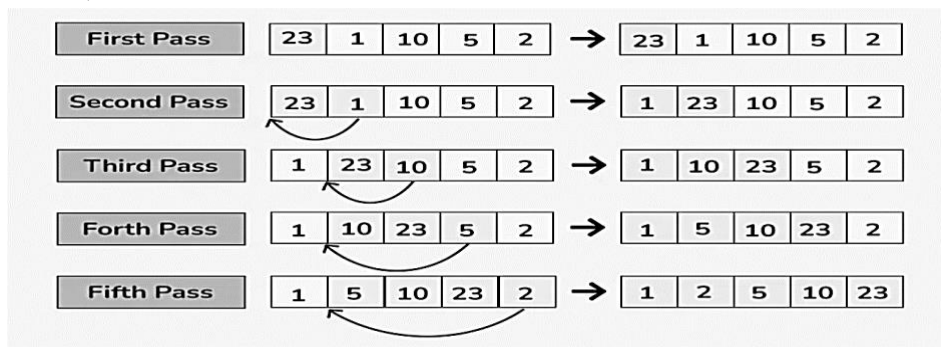
इन्सर्शन सॉर्ट हा एक सोपा सॉर्टिंग अल्गोरिदम आहे जो सॉर्ट न केलेल्या सूचीतील प्रत्येक घटकाला त्याच्या सॉर्ट केलेल्या भागामध्ये त्याच्या योग्य स्थानावर रिकर्षणने समाविष्ट करून कार्य करतो. हे एक स्टॅटिक वर्गीकरण अल्गोरिदम आहे, म्हणजे समान मूल्ये असलेले घटक सॉर्ट केलेल्या आउटपुटमध्ये त्यांचे सापेक्ष क्रम राखतात. इन्सर्शन सॉर्ट करणे म्हणजे आपल्या हातात पत्ते खेळण्यासारखे आहे. तुम्ही कार्डे दोन गटांमध्ये विभागली आहेत: सॉर्ट लावलेली कार्डे आणि न लावलेली कार्डे. त्यानंतर, तुम्ही सॉर्ट न केलेल्या गटातून एक कार्ड निवडा आणि ते सॉर्ट केलेल्या गटामध्ये योग्य ठिकाणी ठेवा.

इन्सर्शन सॉर्ट हा एक सोपा सॉर्टिंग अल्गोरिदम आहे जो एका वेळी सॉर्ट केलेल्या अंर मध्ये एक घटक तयार करून कार्य करतो. याला " इन-प्लेस " सॉर्टिंग अल्गोरिदम मानले जाते, याचा अर्थ मूळ अंरच्या पलीकडे कोणत्याही अतिरिक्त मेमरी स्पेसची आवश्यकता नाही .

इन्सर्शन सॉर्ट साध्य करण्यासाठी, या चरणांचे अनुसरण करा:

1. आपल्याला अंरच्या दुसऱ्या एलिमेंटापासून सुरुवात करावी लागेल कारण अंरमधील पहिला एलिमेंट सॉर्ट केलेला आहे.
2. दुसऱ्या घटकाची पहिल्या घटकाशी तुलना करा आणि दुसरा घटक लहान आहे का ते तपासा, नंतर त्यांची अदलाबदल करा.
3. तिसऱ्या घटकाकडे जा आणि त्याची दुसऱ्या घटकाशी तुलना करा, नंतर प्रथम घटक आणि पहिल्या तीन घटकांमध्ये योग्य स्थितीत ठेवण्यासाठी आवश्यकतेनुसार स्वॅप करा.
4. ही प्रक्रिया सुरू ठेवा, प्रत्येक घटकाची त्याच्या आधीच्या घटकांशी तुलना करा आणि सॉर्ट केलेल्या घटकांमध्ये योग्य स्थितीत ठेवण्यासाठी आवश्यकतेनुसार अदलाबदल करा.
5. संपूर्ण अंर सॉर्ट लावेपर्यंत रिकर्षण करा.

इन्सर्शन सॉर्ट अल्गोरिदमचे कार्य:



पहिला पास:

- वर्तमान घटक 23 आहे
- अंरमधील पहिला घटक सॉर्ट असल्याचे गृहित धरले आहे.
- 0 व्या निर्देशांकापर्यंत सॉर्ट लावलेला भाग आहे: [23]

दुसरा पास:

- 1 ची 23 बरोबर तुलना करा (सॉर्ट केलेल्या भागासह वर्तमान घटक).
- 1 लहान असल्याने, 23 च्या आधी 1 घाला .
- 1ल्या निर्देशांकापर्यंत सॉर्ट लावलेला भाग आहे: [1, 23]

तिसरा पास:

- 1 आणि 23 सह 10 ची तुलना करा (सॉर्ट केलेल्या भागासह वर्तमान घटक).
- 10 1 पेक्षा मोठे आणि 23 पेक्षा लहान असल्याने, 1 आणि 23 मध्ये 10 घाला
- दुसऱ्या निर्देशांकापर्यंत सॉर्ट लावलेला भाग आहे: [1, 10, 23]

चौथी पास:

- 1, 10, आणि 23 सह 5 ची तुलना करा (सॉर्ट केलेल्या भागासह वर्तमान घटक).
- 5 हे 1 पेक्षा मोठे आणि 10 पेक्षा लहान असल्याने 1 आणि 10 मध्ये 5 घाला .
- तिसऱ्या निर्देशांकापर्यंत सॉर्ट लावलेला भाग आहे : [1, 5, 10, 23]

पाचवी पास:

- 2 ची 1, 5, 10 आणि 23 सह तुलना करा (सॉर्ट केलेल्या भागासह वर्तमान घटक).
- 2 हे 1 पेक्षा मोठे आणि 5 पेक्षा लहान असल्याने 1 आणि 5 मध्ये 2 घाला .
- चौथ्या निर्देशांकापर्यंत सॉर्ट लावलेला भाग आहे: [1, 2, 5, 10, 23]

अंतिम अरे:

- सॉर्ट केलेला अरे आहे: [1, 2, 5, 10, 23]

इन्सेरशन सॉर्टची टाइम कॉम्प्लेक्सिटी:

- सर्वोत्तम केस: $O(n)$, जर सूची आधीपासून सॉर्ट केलेली असेल तर, जेथे n ही सूचीमधील घटकांची संख्या आहे.
- सरासरी केस: $O(n^2)$, जर सूची यादृच्छिकपणे क्रमबद्ध केली असेल
- सर्वात वाईट केस: $O(n^2)$, सूची उलट क्रमाने असल्यास

इन्सेरशन सॉर्टची स्पेस कॉम्प्लेक्सिटी:

- ऑक्झिलरी स्पेस: $O(1)$, इन्सर्शन सॉर्टसाठी $O(1)$ अतिरिक्त स्पेस आवश्यक आहे, ज्यामुळे ते एक स्पेस-कार्यक्षम वर्गीकरण अल्गोरिदम बनते.

:Program

```
<include <stdio.h#
<include<conio.h#
Function to perform Insertion Sort //
} void insertionSort(int arr[], int n)
;int i, key, j
} for (i = 1; i < n; i++)
key = arr[i]; // Element to be inserted into the sorted portion
;j = i - 1
Move elements of arr[0..i-1] that are greater than key to one position ahead //
} while (j >= 0 && arr[j] > key)
;arr[j + 1] = arr[j]
;j = j - 1
{
arr[j + 1] = key; // Insert the key into the correct position
{
{
```

```

Function to print the array //
} void printArray(int arr[], int size)
;int i
} for (i = 0; i < size; i++)
;printf("%d ", arr[i])
{
;printf("\n")
{
} ()int main
;int i,n,arr]100[
Input the number of elements //
;printf("Enter the number of elements: ")
;scanf("%d", &n)
Input the array elements //
;printf("Enter %d elements: \n", n)
} for ( i = 0; i < n; i++)
;scanf("%d", &arr[i])
{
;printf("Original array: ")
;printArray(arr, n)
Perform Insertion Sort //
;insertionSort(arr, n)
;printf("Sorted array: ")
;printArray(arr, n)
;()getch
;return 0
{
:Output
Enter the number of elements:4
Enter 4 elements: 4 2 1 7
Sorted array:1 2 4 7

```

3. सिलेक्शन सॉर्ट(Selection Sort)

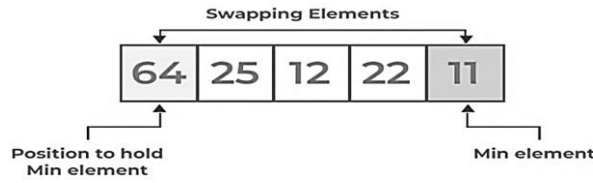
अल्गोरिदम सूचीच्या क्रमबद्ध न केलेल्या भागातून सर्वात लहान घटक वारंवार निवडतो (किंवा सर्वात मोठा) आणि सॉर्ट न केलेल्या भागाच्या पहिल्या घटकासह स्वॅप करतो. संपूर्ण यादी सॉर्ट लावेपर्यंत ही प्रक्रिया उर्वरित अक्रमित भागासाठी रिकर्षण केली जाते. सिलेक्शन सॉर्ट अल्गोरिदम कसे कार्य करते?

उदाहरण म्हणून खालील अॅरेचा विचार करू या: arr[] = {64, 25, 12, 22, 11}

पहिला पास:

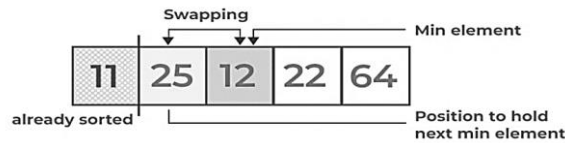
- सॉर्ट करण्यासाठी अॅरेमधील पहिल्या स्थानासाठी, संपूर्ण अॅरे अनुक्रमणिका 0 ते 4 पर्यंत अनुक्रमे पार केली जाते. प्रथम स्थान जेथे 64 सध्या संग्रहित आहे, संपूर्ण अॅरे पार केल्यानंतर हे स्पष्ट होते की 11 हे सर्वात कमी मूल्य आहे.

- अशा प्रकारे, 64 ला 11 ने बदला. एका रिकर्षणनंतर 11 , जे अरेंमध्ये सर्वात कमी मूल्य असेल, सॉर्ट केलेल्या सूचीच्या पहिल्या स्थानावर दिसून येईल.



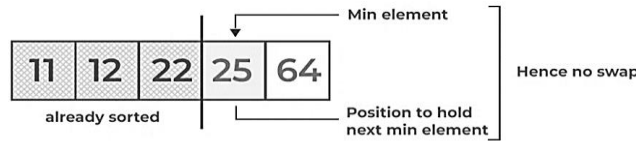
दुसरा पास:

- दुस-या स्थानासाठी, जेथे 25 उपस्थित आहे, पुन्हा अनुक्रमिक पद्धतीने उर्वरित अरें पार करा.
- मार्गक्रमण केल्यावर, आम्हाला आढळले की 12 हे अरेंमधील दुसरे सर्वात कमी मूल्य आहे आणि ते अरेंमध्ये दुसऱ्या स्थानावर दिसले पाहिजे, अशा प्रकारे या मूल्यांची अदलाबदल करा.



तिसरा पास:

- आता, तिसऱ्या स्थानासाठी, जेथे 25 उपस्थित आहे ते पुन्हा उरलेल्या अरेंला मागे टाका आणि अरेंमध्ये उपस्थित असलेले तिसरे सर्वात कमी मूल्य शोधा.
- टॅव्हर्स करताना, 22 हे तिसरे सर्वात कमी मूल्य म्हणून बाहेर आले आणि ते अरेंमध्ये तिसऱ्या स्थानावर दिसले पाहिजे, अशा प्रकारे तिसऱ्या स्थानावर असलेल्या घटकासह 22 स्वॅप करा.



पाचवी पास:

- शेवटी अरेंमध्ये असलेले सर्वात मोठे मूल्य आपोआप अरेंमध्ये शेवटच्या स्थानावर ठेवले जाते. परिणामी अरें हा क्रमबद्ध अरें आहे.



सिलेक्शन सॉर्टचे कॉम्प्लेक्सिटी विश्लेषण

- टाइम कॉम्प्लेक्सिटी:** सिलेक्शन सॉर्टची टाइम कॉम्प्लेक्सिटी $O(N^2)$ आहे कारण दोन नेस्टेड लूप आहेत. अरेंचा एक एक घटक निवडण्यासाठी एक लूप = $O(N)$, त्या घटकाची इतर प्रत्येक अरें घटकाशी तुलना करण्यासाठी दुसरा लूप = $O(N)$.
त्यामुळे एकूणच गुंतागुंत = $O(N) * O(N) = O(N*N) = O(N^2)$
- ऑक्झिलरी स्पेस:** $O(1)$ ही फक्त अतिरिक्त मेमरी म्हणून वापरली जाते ती तात्पुरत्या व्हेरिएबल्ससाठी अरेंमध्ये दोन व्हॅल्यूज स्वॅप करताना. सिलेक्शन सॉर्ट कधीही $O(N)$ पेक्षा जास्त स्वॅप करत नाही आणि जेव्हा मेमरी लेखन महाग असते तेव्हा उपयुक्त ठरू शकते.

Program:

```
#include <stdio.h>
// Function to perform selection sort
void selectionSort(int arr[], int size) {
    int i,j,temp;
```

```

for (i = 0; i < size - 1; i++) {
// Find the minimum element in the unsorted portion of the array
int minIndex = i;
for (j = i + 1; j < size; j++) {
    if (arr[j] < arr[minIndex]) {
        minIndex = j;
    }
}
// Swap the found minimum element with the first unsorted element
if (minIndex != i) {
    temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
}
}
// Function to print the array
void printArray(int arr[], int size) {
int i;
for (i = 0; i < size; i++) {
    printf("%d ", arr[i]);
}
printf("\n");
}
int main() {
int n, arr[100], i;
// Input the number of elements in the array
printf("Enter the number of elements: ");
scanf("%d", &n);
// Input the elements of the array
printf("Enter %d elements: \n", n);
for (i = 0; i < n; i++) {
    scanf("%d", &arr[i]);
}

// Perform the selection sort
selectionSort(arr, n);
// Print the sorted array
printf("Sorted array: \n");
printArray(arr, n);
getch();
return 0;
}

```

Output:

Enter the number of elements:5

Enter 5 elements:6 4 0 2 1

Sorted array:

0 1 2 4 6

4. क्विक सॉर्ट (Quick Sort)

क्विक सॉर्ट हा Divide and Conquer अल्गोरिदमवर आधारित एक सॉर्ट करणारा अल्गोरिदम आहे जो पिव्होट(Pivot) म्हणून एक घटक निवडतो आणि सॉर्ट केलेल्या अंरमध्ये पिव्होटला त्याच्या योग्य स्थितीत ठेवून निवडलेल्या पिव्होटभोवती दिलेल्या अंरचे विभाजन करतो. क्विक सॉर्ट मधील मुख्य प्रक्रिया म्हणजे विभाजन(). विभाजनाचे लक्ष्य म्हणजे पिव्होट (कोणताही घटक पिव्होट म्हणून निवडला जाऊ शकतो) सॉर्ट केलेल्या अंरमध्ये त्याच्या योग्य स्थानावर ठेवणे आणि सर्व लहान घटक पिव्होटच्या डावीकडे आणि सर्व मोठे घटक पिव्होटच्या उजवीकडे ठेवणे हे आहे. पिव्होट योग्य स्थितीत ठेवल्यानंतर पिव्होटच्या प्रत्येक बाजूला विभाजन रिकर्षण केले जाते आणि हे शेवटी अंरची सॉर्ट लावते.

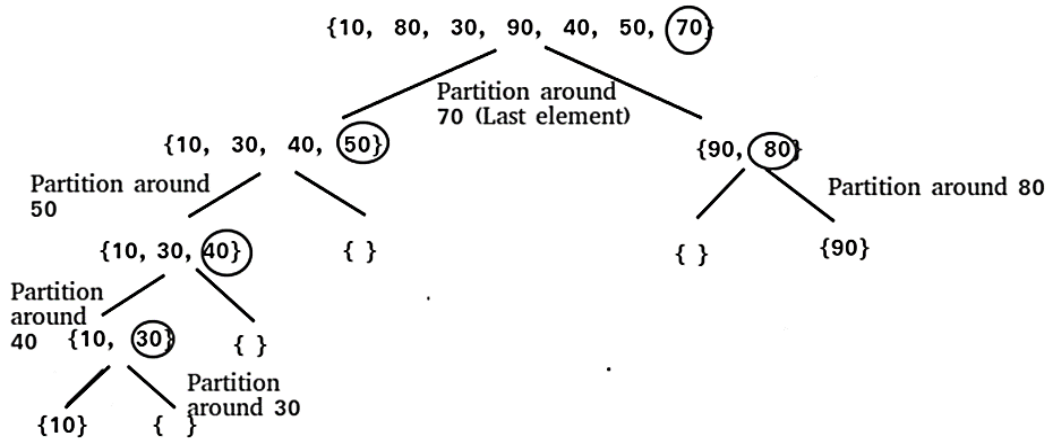


Fig. 2.1. क्विक सॉर्ट (Quick Sort)

पिव्होटची निवड:

पिव्होट्स निवडण्यासाठी बरेच भिन्न पर्याय आहेत.

1. पिव्होट म्हणून पहिला घटक निवडा .
2. शेवटचा घटक मुख्य म्हणून निवडा (खाली लागू)
3. पिव्होट म्हणून एक यादृच्छिक घटक निवडा .
4. पिव्होट म्हणून मध्य निवडा.

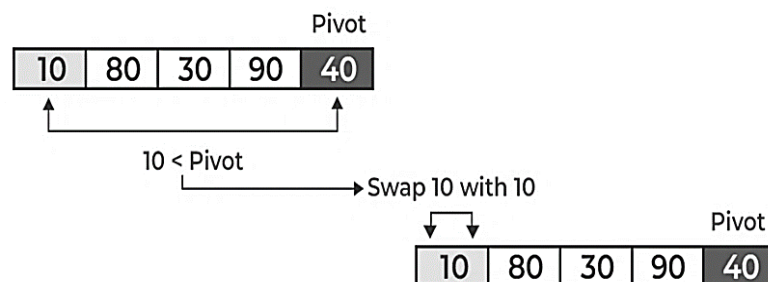
i प्रमाणे लहान (किंवा समान) घटकांच्या निर्देशांकाचा मागोवा ठेवतो . मार्गक्रमण करताना, जर आपल्याला एक लहान घटक आढळला, तर आपण वर्तमान घटक arr[i] सह स्वॅप करतो. अन्यथा, आम्ही वर्तमान घटकाकडे दुर्लक्ष करतो.

क्विक सॉर्टचे कार्य:

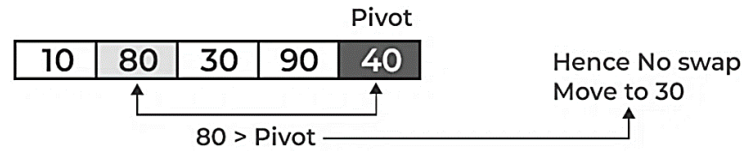
उदाहरण:

विचार करा: arr[] = {10, 80, 30, 90, 40}.

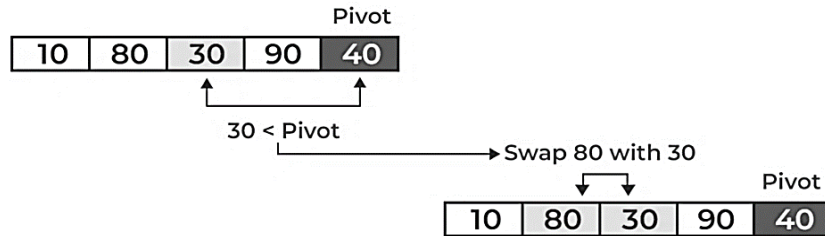
10 ची तुलना पिव्होटशी करा आणि ते पिव्होट पेक्षा कमी असल्यामुळे ते एकंदरीत व्यवस्थित करा.



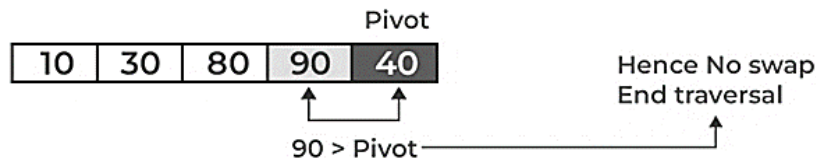
पिव्होटसह 80 ची तुलना करा. ते पिव्होटपेक्षा मोठे आहे.



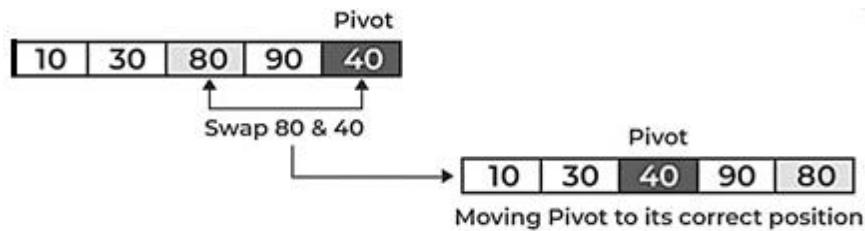
पिव्होटसह 30 ची तुलना करा. ते पिव्होटपेक्षा कमी आहे म्हणून त्यानुसार व्यवस्था करा.



पिव्होटसह 90 ची तुलना करा. ते पिव्होटपेक्षा मोठे आहे.



पिव्होटला त्याच्या योग्य स्थितीत व्यवस्थित करा.



क्विक सॉर्टचे कॉम्प्लेक्सिटी विश्लेषण :

टाइम कॉम्प्लेक्सिटी:

- **सर्वोत्कृष्ट केस (Best case) : $\Omega(N \log N)$**

क्विकसॉर्टसाठी सर्वोत्कृष्ट-केस परिस्थिती उद्भवते जेव्हा प्रत्येक चरणावर निवडलेले पिव्होट अरेला अंदाजे समान भागांमध्ये विभाजित करते. या प्रकरणात, अल्गोरिदम संतुलित विभाजने बनवेल, ज्यामुळे कार्यक्षम वर्गीकरण होईल.

- **सरासरी केस (Average case): $\theta(N \log N)$**

क्विक सॉर्टची सरासरी-केस कामगिरी सहसा सरावात खूप चांगली असते, ज्यामुळे ते सर्वात जलद सॉर्ट करणारा अल्गोरिदम बनते.

- **सर्वात वाईट केस(Worst case): $O(N^2)$**

क्विक सॉर्टची साठी सर्वात वाईट-केस परिस्थिती उद्भवते जेव्हा प्रत्येक स्टेपवरील पिव्होट सातत्याने अत्यंत असंतुलित विभाजनांमध्ये परिणाम करते. जेव्हा अरे आधीपासून सॉर्ट लावलेला असतो आणि पिव्होट नेहमी सर्वात लहान किंवा सर्वात मोठा घटक म्हणून निवडला जातो. सर्वात वाईट परिस्थिती कमी करण्यासाठी, विविध तंत्रे वापरली जातात जसे की एक चांगला पिव्होट निवडणे (उदा. तीनचा मध्य) आणि सॉर्ट लावण्यापूर्वी घटक बदलण्यासाठी यादृच्छिक अल्गोरिदम (यादृच्छिक क्विकसॉर्ट) वापरणे.

स्पेस कॉम्प्लेक्सिटी: $O(1)$, जर आपण रिकर्सिव्ह स्टॅक स्पेसचा विचार केला नाही. जर आपण रिकर्सिव्ह स्टॅक स्पेसचा विचार केला तर, सर्वात वाईट परिस्थितीत क्विकसॉर्ट $O(N)$ बनवू शकतो.

Program:

```
#include <stdio.h>
#include <conio.h>
// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
// Partition function that chooses the pivot and arranges elements
int partition(int arr[], int low, int high) {
    int j, pivot = arr[high]; // Pivot is the last element
    int i = low - 1; // Index of smaller element

    // Rearrange elements based on the pivot
    for (j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++; // Increment the smaller element index
            swap(&arr[i], &arr[j]); // Swap arr[i] and arr[j]
        }
    }
    // Swap the pivot element with the element at i+1
    swap(&arr[i + 1], &arr[high]);
    return i + 1; // Return the pivot index
}
// Function to perform quick sort
void quickSort(int arr[], int low, int high) {
    int pi;
    if (low < high) {
        // Partition the array and get the pivot index
        pi = partition(arr, low, high);
        // Recursively sort the two sub-arrays
        quickSort(arr, low, pi - 1); // Left sub-array
        quickSort(arr, pi + 1, high); // Right sub-array
    }
}
// Function to print the array
void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
}
```

```

printf("\n");
}
int main() {
int n,arr[100],i;
// Input the number of elements in the array
printf("Enter the number of elements: ");
scanf("%d", &n);
// Input the elements of the array
printf("Enter %d elements: \n", n);
for (i = 0; i < n; i++) {
scanf("%d", &arr[i]);
}
// Perform the quick sort
quickSort(arr, 0, n - 1);
// Print the sorted array
printf("Sorted array: \n");
printArray(arr, n);
getch();
return 0;
}

```

Output:

Enter the number of elements:6

Enter 6 elements: 6 4 8 2 1 0

Sorted array:0 1 2 4 6 8

5. मर्ज सॉर्ट (Merge Sort)

मर्ज सॉर्ट हा एक वर्गीकरण अल्गोरिदम आहे जो **विभाजन आणि जिंकण्याच्या** दृष्टिकोनाचे अनुसरण करतो. हे इनपुट अरेला लहान उपअरेमध्ये विभाजीत करून आणि त्या उपअरेची सॉर्ट करून नंतर सॉर्ट केलेला अरे मिळविण्यासाठी पुन्हा एकत्र विलीन करून कार्य करते.

सोप्या भाषेत, आपण असे म्हणू शकतो की विलीनीकरण सॉर्टची प्रक्रिया म्हणजे अरेला दोन भागांमध्ये विभागणे, प्रत्येक अर्धा भागाची सॉर्ट करणे आणि नंतर सॉर्ट केलेले अर्धे उपअरे एकत्र करणे. संपूर्ण अरे सॉर्ट होईपर्यंत ही प्रक्रिया रिकर्षण केली जाते.

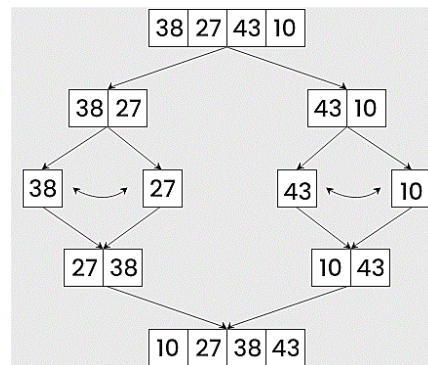


Fig. 2.2. मर्ज सॉर्ट (Merge Sort)

मर्ज सॉर्ट कसे कार्य करते? (How merge sort works)

मर्ज सॉर्ट हे एक लोकप्रिय सॉर्टिंग अल्गोरिदम आहे जे त्याच्या कार्यक्षमता आणि स्टॅटिकतेसाठी ओळखले जाते. घटकांच्या दिलेल्या अरेची सॉर्ट लावण्यासाठी ते विभाजित करा आणि जिंका .

मर्ज सॉर्ट कसे कार्य करते याचे चरण-दर-चरण स्पष्टीकरण येथे आहे:

1. विभाजित करा: सूची किंवा अरे पुन्हा दोन भागांमध्ये विभाजित करा जोपर्यंत ती अधिक विभाजित केली जाऊ शकत नाही.
2. Conquer: प्रत्येक उपअरे subarray मर्ज सॉर्ट अल्गोरिदम वापरून इंडिपेंडेंटली सॉर्ट केला जातो.
3. विलीन करा: सॉर्ट केलेला उपअरे पुन्हा एकत्र केले जातात. दोन्ही सबअरेमधील सर्व घटक विलीन होईपर्यंत प्रक्रिया सुरू राहते.

वरील उदाहरणाचे कार्य पाहू:

विभाजित करा:

- [38, 27, 43, 10] अरे [27, 38] आणि [43, 10] मध्ये विभागलेले आहे .
- [27, 38] उपअरे [38] आणि [27] मध्ये विभागलेले आहे .
- [10, 43] उपअरे [43] आणि [10] मध्ये विभागलेले आहे .

जिंकणे:

- [38] आधीच सॉर्ट केलेला आहे.
- [27] आधीच सॉर्ट केलेला आहे.
- [43] आधीच सॉर्ट केलेला आहे.
- [10] आधीच सॉर्ट केलेला आहे.

विलीन:

- [38 ,27] मिळविण्यासाठी [38] आणि [27] एकत्र करा .
- [43 ,10] मिळविण्यासाठी [43] आणि [10] एकत्र करा .
- [10, 27, 38 , 43] मिळविण्यासाठी [38 ,27] आणि [43 ,10] एकत्र करा
- म्हणून, सॉर्ट केलेली यादी [43 ,38 ,27 ,10] आहे .

विलीनीकरण सॉर्टचे कॉम्प्लेक्सिटी विश्लेषण:**टाइम कॉम्प्लेक्सिटी:**

- **सर्वोत्कृष्ट केस Best case:** $O(n \log n)$, जेव्हा अरे आधीपासून सॉर्ट केलेला असतो किंवा जवळजवळ सॉर्ट केलेला असतो.
- **सरासरी केस Average case:** $O(n \log n)$, जेव्हा अरे यादृच्छिकपणे ऑर्डर केली जाते.
- **सर्वात वाईट केस : (Worst Case)** $O(n \log n)$, जेव्हा अरे उलट क्रमाने सॉर्ट केला जाते.

स्पेस कॉम्प्लेक्सिटी: $O(n)$, विलीनीकरणादरम्यान वापरल्या जाणाऱ्या तात्पुरत्या अरेसाठी अतिरिक्त जागा आवश्यक आहे.

Program:

```
#include <stdio.h>
#include <conio.h>
// Function to merge two halves of the array
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1; // Size of left subarray
    int n2 = right - mid;    // Size of right subarray
    // Temporary arrays for left and right subarrays
    int leftArr[50], rightArr[50], i, j, k;
    // Copy data to temporary arrays
```

```

    for (i = 0; i < n1; i++) {
        leftArr[i] = arr[left + i];
    }
    for (i = 0; i < n2; i++) {
        rightArr[i] = arr[mid + 1 + i];
    }
// Merge the temporary arrays back into the original array
i = 0;
j = 0;
k = left;
while (i < n1 && j < n2) {
    if (leftArr[i] <= rightArr[j]) {
        arr[k] = leftArr[i];
        i++;
    } else {
        arr[k] = rightArr[j];
        j++;
    }
    k++;
}
// Copy the remaining elements of leftArr[], if any
while (i < n1) {
    arr[k] = leftArr[i];
    i++;
    k++;
}

// Copy the remaining elements of rightArr[], if any
while (j < n2) {
    arr[k] = rightArr[j];
    j++;
    k++;
}
}
// Function to implement Merge Sort
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2; // Find the middle point
        // Recursively sort the two halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
// Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

```

```
// Function to print the array
void printArray(int arr[], int size) {
    int i;
    for ( i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: ");
    printArray(arr, n);
    mergeSort(arr, 0, n - 1); // Sorting the array
    printf("Sorted array: ");
    printArray(arr, n);
    getch();
    return 0;
}
```

Output:

Original array: 12 11 13 5 6 7

Sorted array: 5 6 7 11 12 13

संदर्भ (Reference): -

1. Data Structure Using C : Dr. E. Balagurusamy
2. Data structure and Algorithms : K. Mehlhorn
3. Data structure and Algorithms: Dr. Mizanur Rahman
4. Data Structure with c :Lipschutz Schaum Series.
5. <https://www.geeksforgeeks.org/searching-algorithms/>
6. <https://www.geeksforgeeks.org/sorting-algorithms/>

युनिट - 3

लिंकड लिस्ट

(Linked List)

विषय निष्पत्ती (Course Outcome): लिंकड लिस्टवर मूलभूत ऑपरेशन्स अंमलात आणा..

घटक निष्पत्ती (Theory Learning Outcomes):

1. स्टॅटिक (Static) आणि डायनॅमिक (Dynamic) मेमरी वाटप दरम्यान फरक करा.
2. नोडचे रिप्रेझेंट करण्यासाठी लिंकड लिस्ट वापरून योग्य रचना तयार करा.
3. लिनिअर लिंकड लिस्टमधून दिलेला आयटम जोडण्यासाठी किंवा काढण्यासाठी अल्गोरिदम तयार करा.

3.1 मेमरी वाटप (Memory Allocation):- मेमरी वाटप ही एक प्रक्रिया आहे ज्याद्वारे संगणक प्रोग्राम आणि सेवा स्टॅटिक (Static) किंवा डायनॅमिक (Dynamic) मेमरी स्पेससह नियुक्त केल्या जातात. मेमरी वाटप कार्यक्रमाच्या इम्प्लिमेंटेशनच्या आधी किंवा वेळी केले जाते. मेमरी वाटपाचे दोन प्रकार आहेत:

3.1.1 स्टॅटिक मेमरी अॅलोकेशन: (Static Memory Allocation)

स्टॅटिक मेमरी अॅलोकेशन, ज्याला कंपाइल-टाइम मेमरी अॅलोकेशन (Compile Time Memory Allocaton) असेही म्हणतात, एका निश्चित आकारात डेटा संकलित करण्याच्या प्रक्रियेदरम्यान मेमरीच्या वाटपासाठी वापरला जातो. कंपाइलर प्रोग्राममध्ये उपस्थित व्हेरिएबल्ससाठी मेमरी वाटप करतो. हे कार्यक्षम मेमरी प्रवेशास अनुमती देते.

फायदे (Advantages):

1. स्टॅटिक मेमरी अॅलोकेशन व्हेरिएबलला मेमरीचे वाटप करते, जसे की निश्चित अॅरे, ज्याला 'वेळेच्या पुढे' घोषित केले जाते.
2. हे वापरण्यास सोपे आहे.
3. व्हेरिएबल्स कायमस्वरूपी वाटप केले जातात आणि म्हणून, बदलले जाऊ शकत नाहीत ज्यामुळे कार्यप्रदर्शन सुधारते.
4. एक्झिक्यूशन स्पीड टाइम कार्यक्षमतेने नियंत्रित केला जातो आणि जलद मेमरी वाटप आहे.

तोटे (Disadvantages):

1. या पद्धतीमुळे मेमरी अपव्यय होतो.
2. मेमरी आवश्यक नसल्यास, ती मुक्त केली जाऊ शकत नाही. ती कायमस्वरूपी अपरिवर्तित राहते आणि न वापरलेली मेमरी वापरू शकत नाही.

3.1.2 डायनॅमिक मेमरी अॅलोकेशन: (Dynamic Memory Allocation)

डायनॅमिक मेमरी अॅलोकेशन, ज्याला रन-टाइम मेमरी अॅलोकेशन देखील म्हणतात, डेटाच्या इम्प्लिमेंटेशनच्या (रनटाइम) प्रक्रियेदरम्यान मेमरी वाटपासाठी वापरला जातो. जेव्हा प्रोग्राम प्रथमच चालू असेल तेव्हा ते घटकांना वाटप करण्यास अनुमती देते. हे डायनॅमिक सामग्रीसाठी अचूक जागा आणि आकाराचे वाटप करते आणि म्हणूनच, रनटाइम दरम्यान वाटप केले जाते, ज्यामुळे जागेचा अपव्यय कमी होतो.

स्टॅटिक मेमरी अॅलोकेशन (Static Memory Allocation)	डायनॅमिक मेमरी अॅलोकेशन (Dynamic Memory Allocation)
स्टॅटिक मेमरी अॅलोकेशनमध्ये, प्रोग्राम कार्यान्वित होईपर्यंत किंवा फंक्शन कॉल पूर्ण होईपर्यंत व्हेरिएबल्स कायमचे वाटप केले जातात.	डायनॅमिक मेमरी अॅलोकेशनमध्ये, जर तुमचे प्रोग्राम युनिट सक्रिय झाले तरच व्हेरिएबल्सचे वाटप केले जाते.
स्टॅटिक मेमरी अॅलोकेशन प्रोग्रामच्या इम्प्लिमेंटेशनपूर्वी केले जाते.	डायनॅमिक मेमरी वाटप कार्यक्रम इम्प्लिमेंटेशन दरम्यान केले जाते.
हे मेमरीचे स्टॅटिक वाटप व्यवस्थापित करण्यासाठी स्टॅक वापरते	हे मेमरीचे डायनॅमिक वाटप व्यवस्थापित करण्यासाठी रास(Heap) वापरते
ते कमी कार्यक्षम आहे	ते अधिक कार्यक्षम आहे
स्टॅटिक मेमरी अॅलोकेशनमध्ये, मेमरी पुन्हा वापरता येत नाही	डायनॅमिक मेमरी अॅलोकेशनमध्ये, मेमरी पुन्हा वापरता येते आणि आवश्यक नसताना मेमरी मुक्त केली जाऊ शकते

3.2 लिंक केलेल्या सूचीचा परिचय, संज्ञा (Introduction to Linked List and Terminology):-

लिंकड लिस्ट(Linked List): हे डेटा घटकांचे रेखीय संकलन आहे. लिंक केलेल्या सूचीतील प्रत्येक घटकाला 'नोड' असे म्हणतात. प्रत्येक नोडमध्ये दोन फील्ड असतात. प्रथम माहिती (**Information**) माहिती संग्रहित करते दुसरे म्हणजे LINK/NEXT जे यादीतील पुढील नोडच्या ऍड्रेसशी जोडलेले आहे.

उदाहरण:-

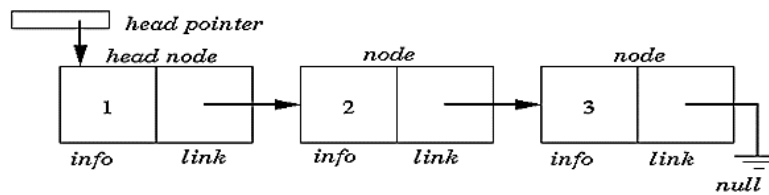


Fig.3.1: लिंकड लिस्ट(Linked List)

3.2.1 संज्ञा (Terminology):-

1. **नोड (Node) :** हे एक डेटा घटक आहे ज्यात माहिती तसेच पुढील नोडचा पत्ता असतो.

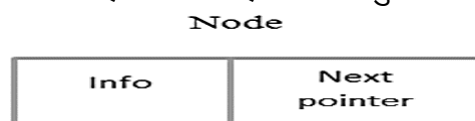


Fig.3.2: नोड (Node)

2. **माहिती (Information /Info):** याला डेटा भाग असेही म्हणतात. हे नोडमध्ये डेटा साठवण्यासाठी वापरले जाते.
3. **लिंकड लिस्टसाठी पुढील पॉइंटर (Next Pointer for Linked list):** जे पुढील नोडचा ऍड्रेस धरतो, जे नोड्सना(nodes) एकत्र जोडते.

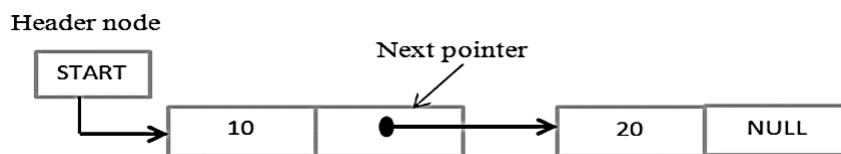


Fig.3.3: नोड (Node)

4. **NULL पॉइंटर (NULL pointer):** याचा वापर लिस्टचा शेवट दर्शवण्यासाठी केला जातो. लिस्ट मधील शेवटचा घटक सूचीचा शेवट दर्शवण्यासाठी NULL सूचक ठेवतो.

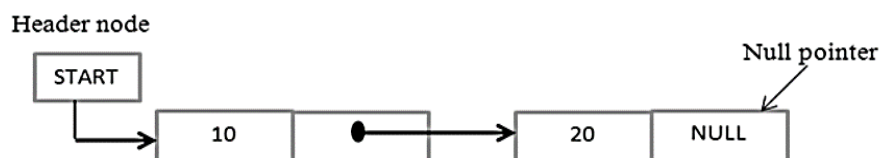


Fig.3.4: NULL पॉइंटर

5. **रिकामी लिस्ट (Empty list):** लिंकड लिस्टला रिकामी म्हणतात जर हेड (प्रारंभ) नोडमध्ये NULL सूचक असेल.



Fig.3.5: रिकामी लिस्ट (Empty list)

6. **ऍड्रेस (Address):** पत्ता नोडचे स्थान सूचित करतो
7. **हेड नोड (Head node):** लिंक केलेल्या सूचीमधील पहिला नोड
8. **हेड पॉइंटर(Head pointer):** - हेड नोडकडे निर्देश करतो

लिंकड लिस्टचे फायदे (Advantages of Linked List):-

1. लिंकड लिस्ट एक गतिमान आहेत जे आवश्यकतेनुसार मेमरी वाटप करतात.
2. समाविष्ट करणे (Addition) आणि हटवणे (Deletion) ऑपरेशन्स सहजपणे लागू केले जाऊ शकतात.
3. स्टॅक (Stack) आणि क्यू (Queue) सहजपणे अंमलात आणल्या जाऊ शकतात.
4. लिंक केलेली यादी प्रवेश वेळ कमी करते.

लिंकड लिस्टचे तोटे (Disadvantages of Link List):-

1. पॉइंटरला स्टोरेजसाठी अतिरिक्त मेमरी आवश्यक असल्याने मेमरी वाया जाते.
2. कोणताही घटक यादृच्छिकपणे प्रवेश केला जाऊ शकत नाही; प्रत्येक नोडमध्ये अनुक्रमे प्रवेश करणे आवश्यक आहे.
3. लिंकड लिस्टमध्ये रिव्हर्स ट्रॅव्हर्सिंग अवघड आहे.

लिंकड लिस्टचे अर्ज (Applications of Link list):-

1. लिंक केलेल्या याद्या स्टॅक, रांगा, आलेख इत्यादी लागू करण्यासाठी वापरल्या जातात.
2. लिंक केलेल्या याद्या तुम्हाला सूचीच्या सुरुवातीला आणि शेवटी घटक घालू देतात.
3. लिंक केलेल्या याद्या मीडिया प्लेयर्समधील प्लेलिस्ट व्यवस्थापित करण्यासाठी वापरल्या जातात, ज्यामुळे गाणी सहज समाविष्ट करणे, हटवणे आणि ट्रॅव्हर्सल करणे शक्य होते.
4. प्रतिमा प्रदर्शित करणारे अनुप्रयोग प्रतिमांचा क्रम व्यवस्थापित करण्यासाठी लिंक केलेल्या सूची वापरू शकतात, पुढे आणि मागे नेव्हिगेशनला अनुमती देतात.

3.2.2 लिंकड लिस्टचे प्रकार (Types of Linked List):-**1. सिंगल लिंकड लिस्ट (Single Linked List)**

सिंगल लिंकड लिस्ट ही मूलभूत डेटा संरचना आहे जी संगणक विज्ञानामध्ये घटकांचा क्रम संग्रहित करण्यासाठी वापरली जाते. यात नोड्सची (Nodes) लिंक (Link) असते, जिथे प्रत्येक नोडमध्ये हे समाविष्ट असते:

- a. **डेटा(Data):** नोडमध्ये असलेले मूल्य किंवा माहिती.
- b. **पॉइंटर(Pointer):** अनुक्रमातील पुढील नोडचा संदर्भ

एकट्या लिंक केलेल्या सूचीवर आम्ही जी ऑपरेशन्स करू शकतो ती

1. समाविष्ट करणे (Addition)
2. हटवणे (Deletion)
3. ट्रॅव्हर्सल (Traversal).

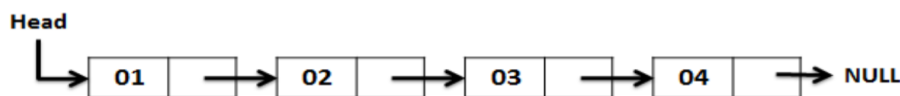


Fig.3.6: सिंगल लिंकड लिस्ट (Single Linked List)

नोड स्ट्रक्चर / रचना (Structure/ Syntax):-

```

#include <stdio.h>
#include <stdlib.h>
// Define the structure for a node
struct Node {
int data;
struct Node* next;
};
  
```

2. सर्क्युलर लिंकड लिस्ट (Circular Linked List):-

सर्क्युलर लिंकड केलेली लिस्ट ही लिंक केलेल्या लिस्टमध्ये एक भिन्नता आहे जिथे शेवटचा नोड पहिल्या नोडकडे निर्देशित करतो आणि वर्तुळ बनवतो. हे काही संदर्भांमध्ये ट्रॅव्हर्सल आणि विशिष्ट ऑपरेशन्स अधिक कार्यक्षम बनवते, कारण शेवटच्या नोडसाठी विशेष प्रकरणे रीसेट किंवा हाताळण्याची गरज न पडता तुम्ही सूचीमधून अनेक वेळा सहजपणे ट्रॅव्हर्सल करू शकता.

सिंगल लिंकड लिस्ट मध्ये केलेल्या सूचीप्रमाणेच सर्क्युलर लिंकड लिस्ट मध्ये केलेल्या सूचीमध्ये नोड्स असतात जिथे प्रत्येक नोडमध्ये हे समाविष्ट असते:

1. **डेटा (Data):** नोडमध्ये असलेले मूल्य किंवा माहिती.
2. **पॉइंटर (Pointer):** अनुक्रमातील पुढील नोडचा संदर्भ.

मुख्य फरक असा आहे की शेवटच्या नोडचा पुढील पॉइंटर हेड नोडकडे निर्देशित करतो, एक गोलाकार रचना तयार करतो.

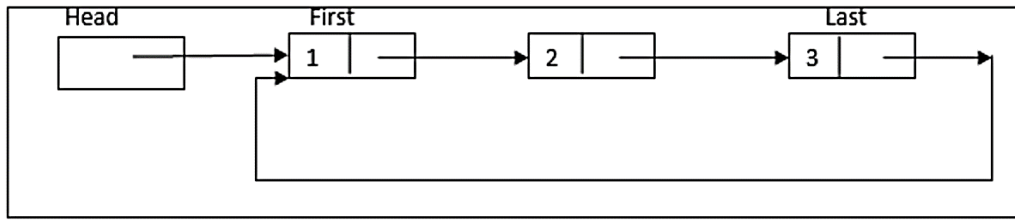


Fig.3.7: सर्क्युलर लिंकड लिस्ट (Circular Linked List)

नोड स्ट्रक्चर / रचना (Structure/ Syntax):-

```
#include <stdio.h>
#include <stdlib.h>
// Define the structure for a node
struct Node {
    int data;
    struct Node* next;
};
```

3. डबली लिंकड लिस्ट (Doubly Linked List):-

डबली लिंकड केलेली लिस्ट ही लिंकड लिस्टचा एक प्रकार आहे ज्यामध्ये प्रत्येक नोडमध्ये तीन फील्ड असतात: एक डेटासाठी आणि दोन पॉइंटर्ससाठी (किंवा संदर्भ) अनुक्रमातील पुढील आणि मागील नोड्ससाठी. हे सिंगल लिंकड केलेल्या लिस्टच्या तुलनेत काही ऑपरेशन्स अधिक कार्यक्षम बनवून, पुढे आणि मागे अशा दोन्ही दिशांनिर्देशांमध्ये लिस्टचा मार्गक्रमण करण्यास अनुमती देते.

डबली लिंकड केलेल्या लिस्टची रचना

डबली लिंकड केलेल्या लिस्टमधील प्रत्येक नोडमध्ये हे आहे:

1. **डेटा (Data):** नोडमध्ये असलेले मूल्य किंवा माहिती.
2. **नेक्स्ट पॉइंटर (Next pointer):** सूचीतील पुढील नोडचा संदर्भ.
3. **प्रीव्हियस पॉइंटर (Previous pointer):** सूचीमधील मागील नोडचा संदर्भ.

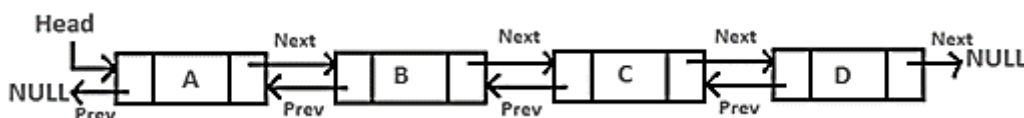


Fig.3.8: डबली लिंकड केलेल्या लिस्टची रचना (Doubly Linked List)

नोड स्ट्रक्चर / रचना (Structure/ Syntax):-

```
#include <stdio.h>
#include <stdlib.h>
// Define the structure for a node
struct Node {
int data;
struct Node* next;
struct Node* prev;
};
```

3.2.3 लिंकड लिस्टवरील ऑपरेशन्स (Operations on Linked List):-

1. **क्रिएट (Create):** हे ऑपरेशन लिंक केलेल्या सूचीमध्ये नोड तयार करण्यासाठी वापरले जाते.
2. **इन्सर्ट (Insert) :** या ऑपरेशनचा वापर लिंक केलेल्या सूचीमध्ये नवीन नोड घालण्यासाठी केला जातो. यादीच्या सुरुवातीला एका विशिष्ट स्थितीत आणि शेवटी.
3. **डिलीट (Delete):** हे ऑपरेशन सूचीमधून नोड हटवण्यासाठी वापरले जाते. यादीच्या सुरुवातीला ,पहिला घटक हटवण्यासाठी ,ठराविक स्थितीत ,शेवटी.
4. **ट्रव्हर्सिंग (Traversing):** ही लिंक केलेल्या सूचीच्या सर्व नोड्समधून एका टोकापासून दुसऱ्या टोकापर्यंत जाण्याची प्रक्रिया आहे.
5. **डिस्प्ले (Display):** हे ऑपरेशन सर्व नोड्स माहिती फील्ड प्रिंट करण्यासाठी वापरले जाते.
6. **शोधणे (Search):** दिलेल्या लिंक केलेल्या सूचीमध्ये विशिष्ट घटक शोधण्यासाठी.
7. **गणना (Count) :** सूचीमध्ये उपस्थित असलेल्या नोड्सची संख्या मोजण्यासाठी.
8. **इरेझ ऑल (Erase all):** सूचीमधून सर्व घटक मुक्त/हटवण्यासाठी .
9. **रिव्हर्स (Reverse):** हे ऑपरेशन नोडचे माहिती फील्ड उलट क्रमाने मुद्रित करण्यासाठी वापरले जाते, म्हणजे शेवटच्या ते पहिल्या पर्यंत.
10. **सॉर्टिंग (Sorting):** संपूर्ण नोडच्या माहिती फील्डची चढत्या किंवा उतरत्या क्रमाने व्यवस्था करणे.

a. सिंगल लिंकड लिस्ट (Single Linked List)मध्ये सुरुवातीला नोड ईन्सर्ट करणे:

1. नवीन नोड तयार करा
2. "डेटा फील्ड" मध्ये डेटा भरा
3. ते "पॉइंटर" किंवा "नेक्स्ट फील्ड" NULL म्हणून बनवा
4. नव्याने तयार केलेला नोड सुरू करण्यासाठी संलग्न करा
5. नवीन नोड प्रारंभी नोड म्हणून बनवा

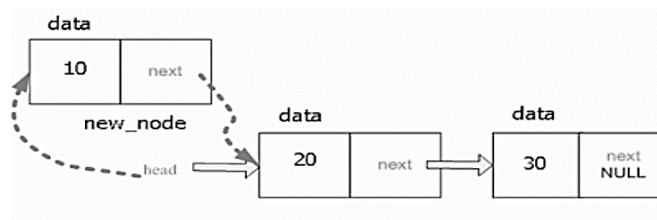


Fig.3.9: नोड ईन्सर्ट करणे (Insertion of node)

b. सिंगल लिंकड लिस्ट (Single Linked List) मध्ये शेवटी नोड ईन्सर्ट करणे:

1. नवीन नोड तयार करा
2. "डेटा फील्ड" मध्ये डेटा भरा
3. ते "पॉइंटर" किंवा "नेक्स्ट फील्ड" NULL बनवा

- नोड लास्ट पोझिशनवर टाकायचे आहे त्यामुळे आम्हाला SLL ला शेवटच्या नोडपर्यंत जावे लागेल.
- शेवटचा नोड आणि नवीन नोड यांच्यात दुवा बनवा

temp -> link = new_node;

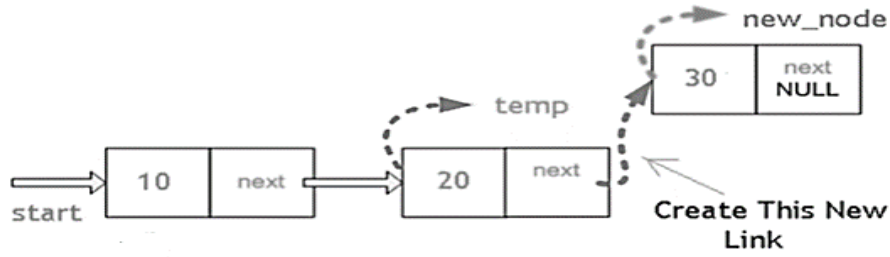


Fig.3.10: नोड ईन्सर्ट करणे (Insertion of node)

सिंगल लिंकड केलेली लिस्ट पार करण्यासाठी अल्गोरिदम

(An algorithm for traversing a singly linked list)

- जर (प्रारंभ == शून्य)
नंतर "लिंक केलेली यादी रिक्त आहे" प्रदर्शित करा.
- अन्यथा लिंक केलेल्या लिस्टच्या प्रत्येक नोडला भेट द्या आणि लिस्टच्या शेवटपर्यंत त्याचा डेटा प्रदर्शित करा
q=start // प्रारंभ नोडला तात्पुरता पॉइंटर q नियुक्त करा
तेव्हा(q!=NULL)
करा
q->डेटा // नोड माहिती प्रदर्शित करा
q=q->लिंक;

सिंगल लिंकड केलेली लिस्टमध्ये नोड शोधण्यासाठी अल्गोरिदम (Algorithms for finding nodes in linked lists)

अल्गोरिदम शोध (माहिती, लिंक, START, आयटम, LOC)

*लिस्ट ही मेमरीमध्ये लिंक केलेली यादी आहे. हे अल्गोरिदम नोडचे स्थान LOC शोधते जेथे ITEM प्रथम LIST मध्ये दिसते किंवा LOC=NULL सेट करते. *

- 1) PTR सेट करा := START
- 2) PTR !=NULL असताना स्टेप 3 ची रिकर्षण करा
- 3) जर ITEM = INFO[PTR], तर
LOC सेट करा: = PTR, आणि बाहेर पडा;
अन्यथा
PTR सेट करा: = LINK [PTR]. (पीटीआर आता पुढील नोडकडे निर्देश करते)
[इफ लूपचा शेवट]
- 4) [शोध अयशस्वी.] सेट LOC:= NULL.
- 5) बाहेर पडा.

सिंगल लिंकड केलेली लिस्टमध्ये नोड्सची संख्या मोजण्यासाठी अल्गोरिदम. (An Algorithm for Counting the Number of Nodes in a Single Link SLL.)

1. प्रारंभ करा
2. यादी रिकामी आहे की नाही ते तपासा
जर phead ==NULL आउटपुट "शून्य नोड्स" आणि बाहेर पडा अन्यथा स्टेप 3 वर जा
3. हेड पॉइंटर तात्पुरत्या व्हेरिएबलवर सेट करा

शेवटचे = phead

4. शेवटच्या नोडपर्यंत मार्गक्रमण करा

SET करताना (last->next != NULL) { नोडची संख्या 1 SET काउंट पॉइंटने यादीतील पुढील नोडपर्यंत वाढवा SET last=last->next }

5. जर सूचीमध्ये एकापेक्षा जास्त नोड असतील तर शेवटच्या नोडसाठी मोजणीचे मूल्य 1 ने वाढवा अन्यथा सूचीमध्ये अगदी एक नोड असल्यास ही अट लागू होईल.

6. नोड्सची एकूण संख्या OUTPUT संख्या प्रदर्शित करा

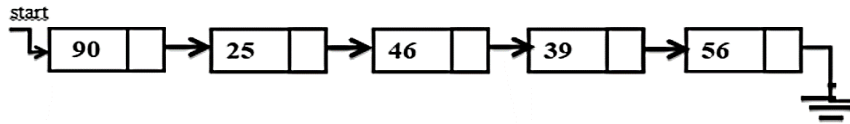
7. थांबा

• डेटा फील्ड 90, 25, 46, 39, 56 वापरून सिंगल लिंकड केलेली लिस्ट तयार करा. SLL वरून नोड 39 शोधा आणि सुरुवातीपासून शेवटपर्यंत आकृतीच्या मदतीने चरण-दर-चरण प्रक्रिया दर्शवा.

(Create a singly linked list using the data fields 90, 25, 46, 39, 56. Find node 39 from SLL and show step by step process with diagram from start to finish.)

सिंगल लिंकड केलेल्या लिस्टमध्ये डेटा फील्ड शोधण्यासाठी, शोध सुरू करणे आवश्यक आहे. सिंगली लिंक केलेल्या लिस्टच्या पहिल्या नोडमधील डेटा फील्ड पुढे दिल्याप्रमाणे.

मूळ सूची:

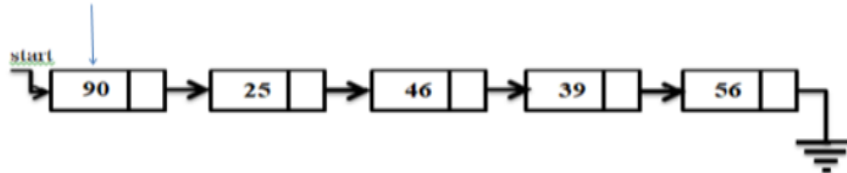


नोड शोधत आहे

1 ली स्टेप:

39 ची 90 शी तुलना करा

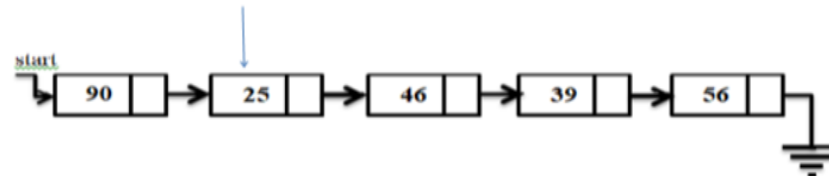
39 != 90,



2 री स्टेप:

39 ची 25 शी तुलना करा

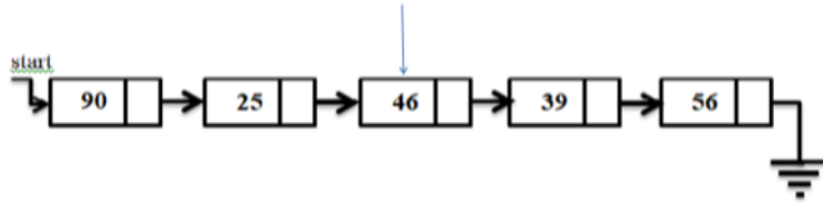
39 != 25,



3 री स्टेप:

39 ची 46 शी तुलना करा

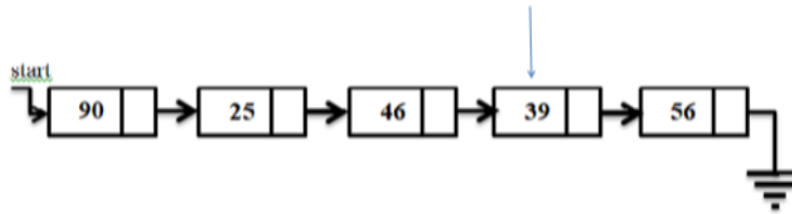
39 != 46,



4 था स्टेप:

39 ची 39 शी तुलना करा

39=39,



नोड 39, 4 वर स्थित आहे. शोध यशस्वी झाला.

SLL मध्ये सुरवातीला एक नवीन नोड घालण्याचा प्रोग्रॅम (A program to insert a new node at the beginning in SLL) :-

```

#include<stdio.h>
#include<conio.h>
#include<malloc.h>
void create_list(int);
void addatbeg(int);
void display();
struct node
{
    int info;
    struct node *link;
} *start;
void main()
{
    int m,n,pos,i;
    clrscr();
    start=NULL;
    printf("how many nodes u want\n");
    scanf("%d",&n);

    for(i=0;i<n;i++)
    {
        printf("enter data value\n");
        scanf("%d",&m);
        create_list(m);
    }
    
```

```

printf("enter data value\n");
scanf("%d",&m);
addatbeg(m);
display(start);
getch();
}
void create_list(int data)
{
    struct node *tmp,*q;
    tmp= malloc(sizeof(struct node));
    tmp->info=data;
    tmp->link=NULL;
    if(start==NULL)
        start=tmp;
    else
    {
        q=start;
        while(q->link!=NULL)
            q=q->link;
        q->link=tmp;
    }
}
void addatbeg(int data)
{
    struct node *tmp;
    tmp= malloc(sizeof(struct node));
    tmp->info=data;
    tmp->link=NULL;
    tmp->link=start;
    start=tmp;
}
void display()
{
    struct node *q;
    if(start==NULL)
    {
        printf("list is empty\n");
    }
    q=start;
    printf("list is:\n");
    while(q!=NULL)
    {
        printf("%d\t",q->info);
        q=q->link;
    }
}

```

```
}
```

संदर्भ (Reference): -

1. Data Structure Using C : Dr. E. Balagurusamy
2. Data structure and Algorithms : K. Mehlhorn
3. Data structure and Algorithms: Dr. Mizanur Rahman
4. Data Structure with c :Lipschutz Schaum Series.

युनिट - 4

स्टॅक

(STACK)

विषय निष्पत्ती (Course Outcome): अर्रे आणि लिंकड लिस्ट इम्प्लिमेंटेशन्स वापरून स्टॅकवर ऑपरेशन्स करा.

घटक निष्पत्ती (Theory Learning Outcomes):

1. अर्रे आणि लिंकड लिस्ट वापरून स्टॅकचे प्रतिनिधित्व करा.
2. स्टॅकमध्ये पुश(PUSH) आणि पॉप (POP) ऑपरेशन्स करण्यासाठी अल्गोरिदम तयार करा..
3. दिलेल्या एक्सप्रेशन इन्फिक्स(Infix) वरून पोस्टफीक्स(Postfix) मध्ये रूपांतरित करण्यासाठी स्टॅक (Stack) वापरा.

4.1 स्टॅकचा परिचय: व्याख्या (Defination):

स्टॅक ही नॉन-प्रिमिटिव्ह रेखीय डेटा संरचना आहे. ही एक ऑर्डर केलेली यादी आहे ज्यामध्ये नवीन डेटा आयटम जोडणे आणि आधीच अस्तित्वात असलेला डेटा आयटम हटवणे हे फक्त एकाच टोकापासून केले जाते , ज्याला **टॉप ऑफ स्टॅक (TOS)** म्हणून ओळखले जाते . स्टॅकमधील सर्व डेटा हटवणे आणि समाविष्ट

करणे स्टॅकच्या शीर्षस्थानी केले जात असल्याने, शेवटचा जोडलेला घटक स्टॅकमधून काढला जाणारा पहिला असेल. या कारणास्तव, स्टॅक देखील म्हणतात **लास्ट-इन-फर्स्ट-आउट (LIFO)** प्रकारची यादी. काही उदाहरणे बघा,

- स्टॅकचे एक सामान्य मॉडेल म्हणजे लग्नाच्या पार्टीत प्लेट्स. ताज्या प्लेट्स वरच्या बाजूला "पुश" केल्या जातात आणि वरच्या बाजूला "पॉप" केल्या जातात.
- तुमच्यापैकी काहीजण बिस्किते खात असतील. कव्हरची फक्त एक बाजू फाटली आहे असे गृहीत धरले तर एक एक करून बिस्किते काढली जातात. याला पॉपिंग म्हणतात आणि त्याचप्रमाणे, जर तुम्हाला काही बिस्किते नंतर काही काळ टिकवून ठेवायची असतील, तर तुम्ही त्यांना पुशिंग नावाच्या फाटलेल्या टोकातून परत पॅकमध्ये ठेवाल.

जेव्हा जेव्हा, स्टॅक तयार केला जातो तेव्हा स्टॅक बेस स्टॅटिक राहतो, जसे की वरून स्टॅकमध्ये नवीन घटक जोडला जातो, स्टॅक टॉप वाढत जातो, उलट स्टॅकचा सर्वात वरचा घटक काढून टाकला जातो तेव्हा स्टॅक टॉप कमी होत जातो.

स्टॅक प्रतिनिधित्व (Stack representation)

1. अरे (स्टॅटिक) वापरणे
2. लिंकलिस्ट वापरणे (डायनॅमिक)

वापर (Application) :

1. फंक्शन्स दरम्यान स्टॅकचा वापर कॉल सुरू होतो
2. डेपथ फर्स्ट सर्चची इम्प्लिमेंटेशन
3. यादी उलटवत आहे
4. कंस तपासणारा
5. इनफिक्स एक्सप्रेसशनचे पोस्टफिक्स एक्सप्रेसशनमध्ये रूपांतर
6. पोस्टफिक्स एक्सप्रेसशनचे मूल्यांकन
7. उपसर्ग एक्सप्रेसशनमध्ये इन्फिक्स एक्सप्रेसशनचे रूपांतर
8. पोस्टफिक्स एक्सप्रेसशनचे मूल्यांकन
9. रिकर्षण
10. हॅनोईचा टॉवर

स्टॅकवरील मूलभूत ऑपरेशन्स (Basic Operation on Stack)

स्टॅक कार्यक्षमतेने वापरण्यासाठी आम्हाला स्टॅकची स्थिती देखील तपासणे आवश्यक आहे. स्टॅक ऑपरेशन्समध्ये स्टॅक सुरू करणे, ते वापरणे आणि नंतर ते डी-इनिशियल करणे समाविष्ट असू शकते.

त्याच उद्देशासाठी, खालील कार्यक्षमता स्टॅकमध्ये जोडली आहे –

- peek() - स्टॅकचा टॉप डेटा घटक न काढता मिळवा.
- isFull() - स्टॅक भरला आहे का ते तपासा.
- isEmpty() - स्टॅक रिक्त आहे का ते तपासा.
- Push() - घटक टाकणे
- Pop() - घटक काढणे
- कोणत्याही वेळी, आम्ही स्टॅकवरील शेवटच्या पुश केलेल्या डेटासाठी पॉइंटर ठेवतो.
- हा पॉइंटर नेहमी स्टॅकच्या वरच्या बाजूस दर्शवतो, म्हणून शीर्ष असे नाव दिले.

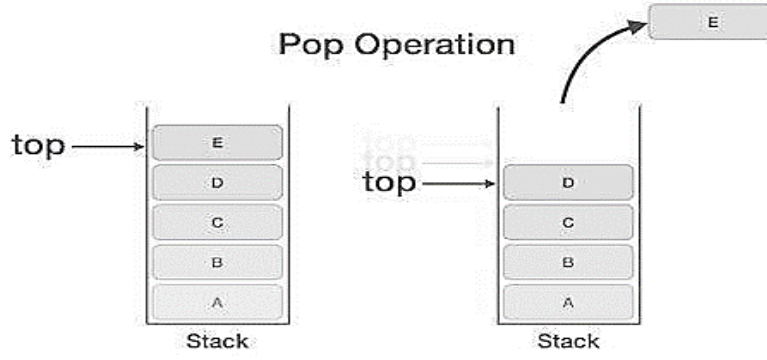


Fig. 4.1: Pop() - घटक काढणे

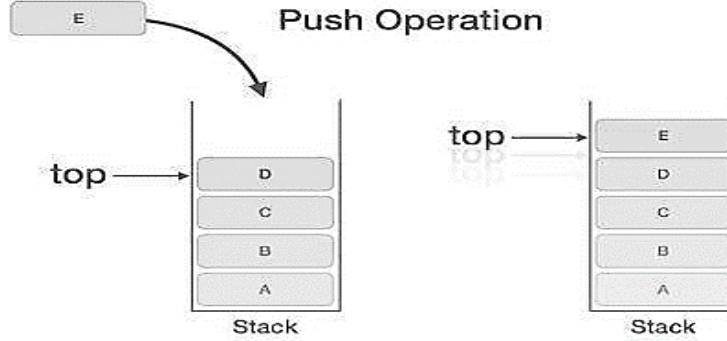


Fig. 4.2: Push() - घटक टाकणे

4.2 स्टॅकची अनुक्रमिक इम्प्लिमेंटेशन

स्टॅक दोन प्रकारे लागू केले जाऊ शकते:

- स्टॅटिक इम्प्लिमेंटेशन (Static Implementation)
- डायनॅमिक इम्प्लिमेंटेशन (Dynamic Implementation)

a. स्टॅटिक इम्प्लिमेंटेशन (Static Implementation)

स्टॅटिक इम्प्लिमेंटेशन स्टॅक तयार करण्यासाठी अर्रे वापरते. स्टॅटिक इम्प्लिमेंटेशन हे एक अतिशय सोपे तंत्र आहे, परंतु निर्मितीचा हा लवचिक मार्ग नाही, कारण प्रोग्राम डिझाइन करताना स्टॅकचा आकार घोषित करावा लागतो, त्यानंतर आकार बदलू शकत नाही. शिवाय, स्टॅटिक इम्प्लिमेंटेशन खूप कार्यक्षम wrt मेमरी वापर नाही. जसे की अर्रेची घोषणा (स्टॅटिक इम्प्लिमेंटेशन करण्यासाठी) ऑपरेशन सुरू होण्यापूर्वी (प्रोग्राम डिझाइन वेळेत) केली जाते, आता स्टॅकमध्ये खूप कमी घटक साठवायचे असल्यास, स्टॅटिकली वाटप केलेली मेमरी वाया जाईल. जर स्टॅकमध्ये जास्त प्रमाणात घटक साठवायचे असतील तर आम्ही अर्रेची क्षमता वाढवण्यासाठी आकार बदलू शकत नाही, जेणेकरून ते नवीन घटक सामावून घेऊ शकतील.

b. डायनॅमिक इम्प्लिमेंटेशन (Dynamic Implementation)

स्टॅटिक इम्प्लिमेंटेशन प्रमाणे, आम्ही स्टॅकमध्ये जोडलेले घटक संचयित करण्यासाठी अर्रे वापरले आहेत. तथापि, अर्रे म्हणून अंमलात आणल्यास त्याला अर्रेच्या मूलभूत मर्यादेचा सामना करावा लागतो- की त्याचा आकार घोषित केलेला आहे तो वाढवता किंवा कमी करता येत नाही. परिणामी, एखाद्या अर्रेसाठी खूप जागा किंवा खूप कमी जागा राखून ठेवते आणि त्याऐवजी स्टॅकसाठी लिंकड लिस्ट वापरून स्टॅक लागू केल्यास या समस्येवर मात करता येईल. लिंक केलेल्या सूचीच्या बाबतीत आम्ही लिंक केलेल्या सूचीच्या एका टोकापासून नोड्स पुश आणि पॉप करू. लिंक केलेली सूची प्रतिनिधित्व सामान्यतः डायनॅमिक इम्प्लिमेंटेशन (Dynamic Implementation) म्हणून ओळखले जाते आणि डेटा संरचना स्टॅक प्रकार लागू करण्यासाठी पॉइंटर्स वापरते. लिंकड लिस्ट म्हणून स्टॅक एकल कनेक्ट केलेली सूची म्हणून दर्शविले जाते. लिंक केलेल्या सूचीतील प्रत्येक

नोडमध्ये डेटा आणि पॉइंटर असतो जो सूचीतील पुढील नोडचे स्थान देतो. यादीतील नोड ही खाली दर्शविल्याप्रमाणे रचना आहे:

```
struct node
{
    <data type> data;
    node *link;
};
```

जिथे <data type> सूचित करते की डेटा कोणत्याही प्रकारचा असू शकतो जसे की int , float, char इत्यादी आणि लिंक, सूचीतील पुढील नोडसाठी एक पॉइंटर आहे. सूचीच्या सुरवातीला पॉइंटर स्टॅकच्या वरच्या भागाचा उद्देश पूर्ण करतो. Fig. 4.3 स्टॅकची लिंक केलेली सूची दर्शवते

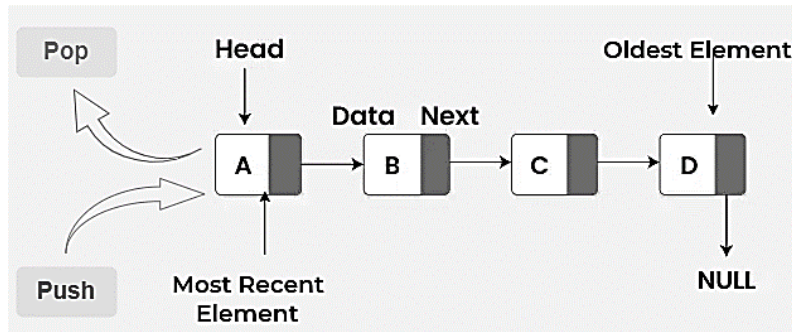


Fig.4.3: लिंकड लिस्ट वापरून स्टॅक (Stack using Linked List)

स्टॅकवर ऑपरेशन्स (Stack Operations)

स्टॅकवर करता येणारी मूलभूत ऑपरेशन्स खालीलप्रमाणे आहेत :

1. **पुश(Push)** : स्टॅकच्या शीर्षस्थानी नवीन घटक जोडण्याच्या प्रक्रियेला पुश ऑपरेशन म्हणतात. स्टॅकमध्ये घटक पुश करण्यामध्ये घटक जोडणे समाविष्ट आहे, कारण नवीन घटक शीर्षस्थानी घातला जाईल, म्हणून प्रत्येक पुश ऑपरेशननंतर, शीर्ष एकाने वाढविला जातो. जर अरे भरलेला असेल आणि कोणताही नवीन घटक सामावून घेता येत नसेल, तर त्याला STACK-FULL स्थिती म्हणतात. या स्थितीला स्टॅक ओव्हरफ्लो म्हणतात .
2. **पॉप(POP)**: स्टॅकच्या वरच्या भागातून एखादा घटक काढण्याचा प्रक्रियेला POP ऑपरेशन म्हणतात. प्रत्येक पॉप ऑपरेशननंतर, स्टॅक एकाने कमी केला जातो. जर स्टॅकवर कोणताही घटक नसेल आणि पॉप केले असेल तर याचा परिणाम स्टॅक अंडरफ्लो स्थितीत होईल

अरे वापरून स्टॅटिक इम्प्लिमेंटेशनसाठी पुश आणि पॉपसाठी अल्गोरिदम (Using Array)

(i) स्टॅकमध्ये आयटम घालण्यासाठी अल्गोरिदम (पुश-PUSH)

STACK[MAXSIZE] हा स्टॅक लागू करण्यासाठी एक अरे आहे, MAXSIZE कमाल दर्शवतो. अरे स्टॅकचा आकार . NUM हा स्टॅकमध्ये ठकलला जाणारा घटक आहे आणि TOP हा स्टॅकच्या शीर्षस्थानी असलेल्या घटकाचा अनुक्रमणिका क्रमांक आहे.

स्टेप 1 : [स्टॅक ओव्हरफ्लो तपासा?]

TOP = MAXSIZE – 1 असल्यास, नंतर:

लिहा : 'स्टॅक ओव्हरफ्लो' आणि परत या.

[इफ स्ट्रक्चरचा शेवट]

स्टेप 2 : स्टॅकमध्ये ठकलण्यासाठी NUM वाचा.

स्टेप 3 : TOP = TOP + 1 सेट करा [TOP 1 ने वाढवते]

स्टेप 4 : STACK[TOP] = NUM सेट करा [नवीन क्रमांक NUM नवीन टॉप पोजिशनमध्ये घालते]

स्टेप 5: बाहेर पडा

C language मधील स्टॅक पुश ऑपरेशनचे कार्य खालीलप्रमाणे आहे

```
void push()
{
    if(top==MAXSIZE-1)
    {
        printf("\n\nStack is full(Stack overflow)"); return;
    }
    int num;
    printf("\n\nEnter the element to be pushed in stack : "); scanf("%d",&num);
    top++;
    stack[top]=num;
}
```

(ii) स्टॅकमधून आयटम काढण्यासाठी अल्गोरिदम (POP)

STACK[MAXSIZE] हा स्टॅक लागू करण्यासाठी एक अर्रे आहे जेथे MAXSIZE कमाल दर्शवते. अर्रे स्टॅकचा आकार. NUM हा स्टॅकमधून पॉप केलेला घटक आहे आणि TOP हा स्टॅकच्या शीर्षस्थानी असलेल्या घटकाचा अनुक्रमणिका क्रमांक आहे.

स्टेप 1 : [स्टॅक अंडरफ्लोसाठी तपासा?]

जर TOP = - 1 : तर

लिहा : 'स्टॅक अंडरफ्लो' आणि परत या.

[इफ स्ट्रक्चरचा शेवट]

स्टेप 2 : NUM = STACK[TOP] सेट करा [NUM ला शीर्ष घटक नियुक्त करा]

स्टेप 3 : 'स्टॅकमधून पॉप केलेला घटक आहे :', NUM लिहा.

स्टेप 4 : TOP = TOP - 1 सेट करा [शीर्ष 1 ने कमी करते]

स्टेप 5: बाहेर पडा

C language मधील स्टॅक पॉप ऑपरेशनचे कार्य खालीलप्रमाणे आहे :

```
void pop()
{
    if(top== -1)
    {
        printf("\n\nStack is empty(Stack underflow)"); return;
    }
    int num; num=stack[top];
    printf("\n\nElement popped from stack : %d",num);
}
```

Program 1 : Static implementation of stacks using arrays

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAXSIZE 5
void push();
void pop();
```

```

void display();
int stack[MAXSIZE];
int top=-1;
void main()
{
clrscr(); int choice;
while(1)
{
clrscr();
printf("STATIC STACK");
printf("\n-----");
printf("\n1. PUSH");
printf("\n2. POP");
printf("\n3. DISPLAY");
printf("\n4. EXIT");
printf("\n-----");
printf("\n\nEnter your choice [1/2/3/4] : "); scanf("%d",&choice);
switch(choice)
{
case 1 :
push();
break;
case 2 :
pop();
break;
case 3 :
display();
break;
case 4 :
exit(0);
default :
printf("\n\nInvalid choice");
}
getch();
}
}

// Function for the operation push
void push()
{
int num; if(top==MAXSIZE-1)
{
printf("\n\nStack is full(Stack overflow)");
return;
}
printf("\n\nEnter the element to be pushed in stack : ");

```

```
scanf("%d",&num);
top++;
stack[top]=num;
}
// Function for the operation pop
void pop()
{
int num; if(top==1)
{
printf("\n\nStack is empty(Stack underflow)");
return;
}
num=stack[top];
printf("\n\nElement popped from stack : %d",num);
top--;
}
// Function for traversing the stack
void display()
{
if(top==1)
{
printf("\n\nStack is empty(Stack underflow)"); return;
}
printf("\n\nStack elements are : \n");
for(int i=top;i>=0;i--)
printf("Stack[%d] : %d\n",i,stack[i]);
}
```

पॉइंटर वापरून डायनॅमिक इम्प्लिमेंटेशनसाठी पुश आणि पॉपसाठी अल्गोरिदम (Using Pointer)

(i) स्टॅकमध्ये आयटम घालण्यासाठी अल्गोरिदम (पुश)

PTR हा स्ट्रक्चर पॉइंटर आहे जो नवीन नोडसाठी मेमरी वाटप करतो आणि NUM हा घटक स्टॅकमध्ये ढकलला जातो, TOP स्टॅकच्या शीर्षस्थानी नोडचा पत्ता दर्शवतो, INFO नोडच्या माहितीचा भाग दर्शवतो आणि LINK लिंकचे प्रतिनिधित्व करतो. किंवा पुढील पॉइंटर पुढील नोडच्या पत्त्याकडे निर्देशित करतो.

स्टेप 1: PTR वापरून नवीन नोडसाठी मेमरी वाटप करा.

स्टेप 2 : स्टॅकमध्ये ढकलण्यासाठी NUM वाचा.

स्टेप 3 : PTR->INFO = NUM सेट करा

स्टेप 4 : PTR->LINK=TOP सेट करा

स्टेप 5 : TOP = PTR सेट करा

स्टेप 6: बाहेर पडा

पुशसाठी कोड (Code for PUSH)

```
void push()
{
struct stack *ptr;
```

```
int num;
ptr=(struct stack *)malloc(sizeof(struct stack)); printf("\nEnter
the element to be pushed in stack : ");
scanf("%d",&num); ptr-
>info=num;
ptr->link=top;
top=ptr;
}
```

(ii) स्टॅकमधून घटक काढण्यासाठी अल्गोरिदम (POP)

चला पीटीआर हा स्ट्रक्चर पॉइंटर आहे जो स्टॅकच्या शीर्षस्थानी नोडची मेमरी डिलॉक करतो आणि NUM हा घटक स्टॅकमधून पॉप केला जातो, TOP स्टॅकच्या शीर्षस्थानी नोडचा पत्ता दर्शवतो, INFO नोडचा माहिती भाग दर्शवतो आणि LINK पुढील नोडच्या पत्त्याकडे निर्देशित करणारा दुवा किंवा पुढील पॉइंटर दर्शवते.

स्टेप 1 : [स्टॅक अंडरफ्लो तपासा?]

जर TOP = NULL : तर

'स्टॅक अंडरफ्लो' लिहा आणि परत या. [इफ स्ट्रक्चरचा शेवट]

स्टेप 2 : PTR=TOP सेट करा.

स्टेप 3 : NUM=PTR->माहिती सेट करा

स्टेप 4 : 'स्टॅकमधून पॉप केलेले घटक is :', NUM लिहा

स्टेप 5 : TOP=TOP->पुढील सेट करा

स्टेप 6 : PTR वापरून शीर्षस्थानी नोडची मेमरी डिलॉक करा.

स्टेप 5: बाहेर पडा

POP साठी कोड (Code for POP)

```
void pop()
{
if(top==NULL)
{
printf("\nStack is empty(Stack underflow).");
return;
}
struct stack *ptr; int num;
ptr=top;
num=ptr->info;
printf("\nElement popped from stack : %d",num);
top=top->link;
free(ptr);
}
```

Program 2 : Dynamic implementation of stacks using pointers

प्रोग्राम २ : पॉइंटर वापरून स्टॅकची डायनॅमिक इम्प्लिमेंटेशन

```
#include<stdio.h>
#include<conio.h> #include<stdlib.h>
struct stack
```

```
{
int info;
struct stack *link;
}*top=NULL;
void push();
void pop();
void display();
void main()
{
int choice;
while(1)
{
clrscr();
printf("DYNAMIC IMPLEMENTATION OF STACKS");
printf("\n-----");
printf("\n1. PUSH");
printf("\n2. POP");
printf("\n3. DISPLAY");
printf("\n4. EXIT");
printf("\n-----");
printf("\nEnter your choice [1/2/3/4] : ");
scanf("%d",&choice);
switch(choice)
{
case 1 : push();
break;
case 2 : pop();    break;
case 3 : display();
break;
case 4 : exit(0);
default:
printf("\nInvalid choice.");
}
getch();
}
}

// Function for the push operation
void push()
{
struct stack *ptr; int num;
ptr=(struct stack *)malloc(sizeof(struct stack));
printf("\nEnter the element to be pushed in stack : ");
scanf("%d",&num); ptr->info=num;
ptr->link=top;
top=ptr;
```

```

}
// Function for the pop operation
void pop()
{
    struct stack *ptr; int num; ptr=top;
    if(top==NULL)
    {
        printf("\nStack is empty(Stack underflow).");
        return;
    }
    num=ptr->info;
    printf("\nElement popped from stack : %d",num);
    top=top->link;
    free(ptr);
}
// Function for traversing the stack
void display()
{
    struct stack *ptr; ptr=top;
    if(top==NULL)
    {
        printf("\nStack is empty(Stack underflow).");
        return;
    }
    printf("\nThe elements of stack are :\n");
    while(ptr!=NULL)
    {
        printf("%d\n",ptr->info); ptr=ptr->link;
    }
}

```

4.3 पोलिश-नोटेशन (POLISH-NOTATIONS)

अंकगणितीय एक्सप्रेशनचे (Arithmetic expression) मूल्यमापन करण्यासाठी ज्या ठिकाणी स्टॅकचा वारंवार वापर केला जातो. अंकगणित एक्सप्रेशनमध्ये ऑपरेंड आणि ऑपरेटर असतात. ऑपरेंड संख्यात्मक मूल्ये किंवा संख्यात्मक चल असू शकतात. वापरलेले ऑपरेटर एक अंकगणितीय एक्सप्रेशन आहे जे बेरीज, वजाबाकी, गुणाकार, भागाकार आणि घातांक या क्रियांचे प्रतिनिधित्व करतात .

जेव्हा उच्च स्तरीय प्रोग्रामिंग भाषा अस्तित्वात आल्या तेव्हा संगणक शास्त्रज्ञांसमोरील प्रमुख अडथळ्यांपैकी एक म्हणजे मशीन भाषेच्या सूचना व्युत्पन्न करणे जे कोणत्याही अंकगणितीय एक्सप्रेशनचे योग्यरित्या मूल्यांकन करेल. $X = A / B + C * D - F * G / Q$ सारखे जटिल असाइनमेंट स्टेटमेंट रूपांतरित करण्यासाठी

योग्य सूचना क्रम मध्ये एक कठीण काम होते. एक्सप्रेशनच्या मूल्यमापनाचा क्रम निश्चित करण्यासाठी प्रत्येक भाषा प्रत्येक ऑपरेटरला प्राधान्य देते.

ए पोलिश (A.Polish) गणितज्ञाने पोलिश नोटेशन नावाचे नोटेशन सुचवले, जे अंकगणित एक्सप्रेशनचे प्रतिनिधित्व करण्यासाठी दोन पर्याय देते. नोटेशनस उपसर्ग आणि पोस्टफिक्स नोटेशनस आहेत. पोलिश नोटेशनचा मूलभूत गुणधर्म असा आहे की ऑपरेशनस ज्या क्रमाने करायच्या आहेत ते पूर्णपणे एक्सप्रेशनमधील ऑपरेटर आणि

ऑपरेंडच्या स्थानांवरून निर्धारित केले जातात. म्हणून पोलिश नोटेशनमध्ये एक्सप्रेशन लिहिताना कंस आवश्यक नाही. पोलिश नोटेशनचे दोन प्रकार खाली दिले आहेत:

1. **प्रीफिक्स नोटेशन (Prefix Notation)**
2. **पोस्टफिक्स नोटेशन (Postfix Notation)**
3. **इन्फिक्स नोटेशन (Infix Notation)**

1. प्रीफिक्स नोटेशन: (Prefix Notation)

प्रीफिक्स नोटेशन एक नोटेशन आहे ज्यामध्ये ऑपरेटर ऑपरेंड्सच्या आधी लिहिलेला असतो. उदाहरणार्थ,
 $+ AB$

ऑपरेटर '+' ऑपरेंड्स A आणि B च्या आधी लिहिलेला असल्याने, या नोटेशनला प्रीफिक्स नोटेशन (पूर्व म्हणजे आधी) असे म्हणतात.

2. पोस्टफिक्स नोटेशन: (Postfix Notation)

पोस्टफिक्स नोटेशन एक नोटेशन आहे ज्यामध्ये ऑपरेटर ऑपरेंड्स नंतर लिहिलेला असतो. उदाहरणार्थ,
 $AB +$

ऑपरेटर '+' हे ऑपरेंड्स A आणि B नंतर लिहिलेले असल्यामुळे या नोटेशनला पोस्टफिक्स नोटेशन (पोस्ट म्हणजे नंतर) असे म्हणतात.

3. इन्फिक्स नोटेशन: (Infix Notation)

इन्फिक्स नोटेशन हे आपल्या सामान्य गणितामध्ये आढळते, जिथे ऑपरेटर ऑपरेंड्समध्ये लिहिलेला असतो. उदाहरणार्थ: दोन संख्या A आणि B जोडण्यासाठी एक्सप्रेशन इन्फिक्स नोटेशनमध्ये लिहिली आहे:

$$A + B$$

लक्षात घ्या की ऑपरेटर '+' ऑपरेंड्स A आणि B मध्ये लिहिलेला आहे. या नोटेशनला infix असे का म्हटले जाते, ते एक्सप्रेशनमध्ये ऑपरेटरचे स्थान आहे.

नोटेशन रूपांतरणे (Conversion of Notation)

$A + B * C$ ही एक्सप्रेशन दिली आहे, जी इन्फिक्स नोटेशनमध्ये आहे. A, B, C साठी अनुक्रमे 4, 3, 7 या मूल्यांसाठी या एक्सप्रेशनची गणना करण्यासाठी, योग्य निकाल मिळविण्यासाठी आपण काही नियम (सामान्य गणितात BODMAS म्हणतात) पाळले पाहिजेत. उदाहरणार्थ,

$$A + B * C = 4 + 3 * 7 = 7 * 7 = 49$$

हा निकाल योग्य नाही कारण बेरीज करण्यापूर्वी गुणाकार करावयाचा आहे कारण त्याला बेरीजपेक्षा जास्त प्राधान्य आहे. याचा अर्थ असा की एक्सप्रेशनची गणना करण्यासाठी आपल्याला ऑपरेटरच्या अग्रक्रमाचे ज्ञान असणे आवश्यक आहे.

ऑपरेटर प्राधान्य (Prefernce of operator)

एक्सपोनेन्शियल ऑपरेटर ^ सर्वोच्च प्राधान्य

गुणाकार/भागाकार *, / पुढील अग्रक्रम

बेरीज/वजाबाकी +, - किमान प्राधान्य

इन्फिक्स एक्सप्रेशन पोस्टफिक्स एक्सप्रेशनमध्ये रूपांतरित करणे (infix to postfix)

$A + B * C$ इन्फिक्स फॉर्म

$A + (B * C)$

$A + (BC*)$ गुणाकार रूपांतरित करा

$A(BC*)+$ बेरीज रूपांतरित करा

$ABC*+$ पोस्टफिक्स फॉर्म

इनफिक्सला पोस्टफिक्स एक्सप्रेसनमध्ये रूपांतरित करण्याचे नियम (Rules):

1. डावीकडून उजवीकडे सुरू होणारी एक्सप्रेसन कंस करा.
2. एक्सप्रेसनचे कंस करताना, उच्च प्राधान्य असलेल्या ऑपरेटरशी संबंधित ऑपरेण्ड्स प्रथम कंसात आकारले जातात. उदाहरणार्थ वरील एक्सप्रेसन $B * C$ मध्ये प्रथम $A + B$ च्या आधी कंस केला आहे.
3. उप-एक्सप्रेसन (एक्सप्रेसनचा भाग) जे पोस्टफिक्समध्ये रूपांतरित केले गेले आहे ते सिंगल ऑपरेण्ड मानले जाईल.
4. एकदा एक्सप्रेसन पोस्टफिक्स फॉर्ममध्ये रूपांतरित झाल्यावर कंस काढून टाका.

इनफिक्स एक्सप्रेसन पोस्टफिक्स फॉर्ममध्ये रूपांतरित करण्याची उदाहरणे:**1. खालील एक्सप्रेसनचे पोस्टफिक्स मध्ये रूपांतर करा**

$$A + B - C$$

उत्तर:

$$(A + B) - C$$

$$(AB +) - C$$

$$\text{चला } T = (AB +) \quad \text{RULE}$$

$$T - C$$

$$TC-$$

किंवा

$$AB + C - \quad \text{पोस्टफिक्स एक्सप्रेसन}$$

2. खालील एक्सप्रेसनचे पोस्टफिक्स मध्ये रूपांतर करा

$$A * B + C$$

उत्तर:

$$(A * B) + C$$

$$(AB *) + C$$

$$\text{चला } T = (AB *)$$

$$T + C$$

$$TC +$$

किंवा

$$AB * C + \quad \text{पोस्टफिक्स एक्सप्रेसन}$$

3. खालील एक्सप्रेसनचे पोस्टफिक्स मध्ये रूपांतर करा

$$A * B + C/D$$

उत्तर:

$$(A * B) + C/D$$

$$(AB *) + C/D$$

$$\text{चला } T = (AB *)$$

$$T + C/D$$

$$T + (C/D)$$

$$T + (CD /)$$

$$\text{चला } S = (CD /)$$

$$T + S$$

$$TS +$$

$$\text{किंवा } AB * CD / + \quad \text{पोस्टफिक्स एक्सप्रेसन}$$

4. खालील एक्सप्रेशनचे पोस्टफिक्स मध्ये रूपांतर करा

$$A + B/C - D$$

उत्तर:

$$(A + (B/C) - D$$

$$A + (BC/) - D$$

$$\text{चला } T = (BC /)$$

$$A + T - D$$

$$(A + T) - D$$

$$(AT +) - D$$

$$\text{चला } S = (AT +)$$

$$S-D$$

$$SD -$$

$$\text{किंवा } AT + D -$$

$$\text{किंवा } ABC / + D - \quad \text{पोस्टफिक्स एक्सप्रेशन}$$

5. खालील एक्सप्रेशनचे पोस्टफिक्स मध्ये रूपांतर करा

$$(A + B)/(C - D)$$

उत्तर:

$$(A + B)/(C - D)$$

$$(AB +)/(C - D)$$

$$(AB +)/(CD -)$$

$$\text{चला } T = (AB +) \text{ \& } S = (CD -)$$

$$T/S$$

$$TS-/$$

$$AB + CD - / \quad \text{पोस्टफिक्स एक्सप्रेशन}$$

6. खालील एक्सप्रेशनचे पोस्टफिक्स मध्ये रूपांतर करा

$$(A + B) * C/D$$

उत्तर:

$$(A + B) * C/D$$

$$(AB +) * C/D$$

$$\text{चला } T = (AB +)$$

$$(T * C)/D$$

$$(TC *)/D$$

$$\text{चला } S = (TC *)$$

$$S/D$$

$$SD /$$

$$TC * D /$$

$$AB + C * D/ \quad \text{पोस्टफिक्स एक्सप्रेशन}$$

7. खालील एक्सप्रेशनचे पोस्टफिक्स मध्ये रूपांतर करा

$$(A + B) * C/D + E \wedge F/G$$

उत्तर:

$$(AB +) * C/D + E \wedge F / G$$

$$\text{चला } T = (AB +)$$

$$T * C/D + (E^F) / G$$

$$T * C/D + (EF \wedge) / G$$

चला $S = (EF \wedge)$

$$T * C/D + S/G$$

$$(T * C)/D + S/G$$

$$(TC *)/D + S/G$$

चला $Q = (TC *)$

$$Q/D + S/G$$

$$(Q/D) + S/G$$

$$(QD /) + S/G$$

चला $P = (QD /)$

$$P + S/G$$

$$P + (S/G)$$

$$P + (SG /)$$

$O = (SG /)$ द्या

$$P + O$$

$$PO +$$

आता आपण $PO +$ ही एक्सप्रेसन विस्तृत करू

$$PO +$$

$$PSG / +$$

$$QD/SG / +$$

$$TC * D / SG / +$$

$$TC * D / EF \wedge G / +$$

$$AB + C * D / EF \wedge G / + \quad \text{पोस्टफिक्स एक्सप्रेसन}$$

8. खालील एक्सप्रेसनचे पोस्टफिक्स मध्ये रूपांतर करा

$$[(B + C) + (D + E) * F] / G$$

उत्तर:

$$(B + [(B + C) + (D + E) * F] / G$$

$$A + [(BC +) + (DE +) * F] / G$$

चला $T = (BC +)$ & $S = (DE +)$

$$A + [T + S * F] / G$$

$$A + [T + (SF *)] / G$$

चला $Q = (SF *)$

$$A + [T + Q] / G$$

$$A + (TQ +) / G$$

चला $P = (TQ +)$

$$A + P / G$$

$$A + (PG /)$$

चला $N = (PG /)$

$$A + N$$

$$AN +$$

$AN +$ या एक्सप्रेसन विस्तार केल्याने मिळते

$$APG / +$$

ATQ + G / +

ATSF * + G / +

ABC + DE + F * + G / + पोस्टफिक्स एक्सप्रेशन

9. खालील एक्सप्रेशनचे पोस्टफिक्स मध्ये रूपांतर करा

$$A + (B * C - (D / E \wedge F) * G) * H$$

उत्तर:

$$(C + (B * C - (D / E \wedge F) * G) * H$$

$$A + (B * C - (D / (EF \wedge)) * G) * H$$

T = (EF ^) द्या

$$A + (B * C - (D / T) * G) * H$$

$$A + (B * C - (DT /) * G) * H$$

चला S = (DT /)

$$A + (B * C - S * G) * H$$

$$A + (B * C - (SG *)) * H$$

चला Q = (SG *)

$$A + (B * C - Q) * H$$

$$A + (B * C) - Q) * H$$

$$A + (BC *) - Q) * H$$

$$P = (BC *) द्या$$

$$A + (P - Q) * H$$

$$A + (PQ -) * H$$

चला O = (PQ -)

$$A + O * H$$

$$A + (OH *)$$

चला N = (OH *)

$$A + N$$

AN +

AN + या एक्सप्रेशनचा विस्तार केल्यास,

$$AOH * +$$

$$APQ - H * +$$

$$ABC * Q - H * +$$

$$ABC * SG * - H * +$$

$$ABC * DT / G * - H * +$$

ABC * DEF ^ / G * - H * + पोस्टफिक्स एक्सप्रेशन

10. खालील एक्सप्रेशनचे पोस्टफिक्स मध्ये रूपांतर करा

$$A - B / (C * D \wedge E).$$

उत्तर:

$$(D - B / (C * D \wedge E)$$

$$A - B / (C * (DE \wedge))$$

T = (DE ^) द्या

$$A - B / (C * T)$$

$$A - B / (CT *)$$

चला $S = (CT *)$

$A - B/S$

$A - (BS /)$

चला $Q = (BS /)$

$A - Q$

$AQ -$

आता AQ या एक्सप्रेशनचा विस्तार करत आहे -

$AQ -$

$ABS / - ABCT *$

$/ -$

$ABCDE \wedge * / -$ पोस्टफिक्स एक्सप्रेशन

• इन्फिक्स एक्सप्रेशन्स पोस्टफिक्स एक्सप्रेशन्समध्ये रूपांतरित करणे

- ऑपरेंड्स नेहमी एकमेकांच्या संदर्भात समान क्रमाने राहतात
- ऑपरेटर फक्त ऑपरेंडच्या उजवीकडे जातात
- यापुढे गरज नसतानाच कंस काढले जातात

• स्ट्रिंग postfixExp = null;

• भेटली तर

- infixExp स्ट्रिंगवरील ऑपरेंड , ते postfixExp मध्ये जोडा
- "(" infixExp स्ट्रिंगवर, स्टॅकवर ढकलून द्या
- infixExp स्ट्रिंगवरील ऑपरेटर ,
 - स्टॅक रिकामा असल्यास, ऑपरेटरला स्टॅकवर ढकलून द्या
 - स्टॅक रिक्त नसल्यास, स्टॅकमधून अधिक किंवा समान प्राधान्य असलेले पॉप ऑपरेटर आणि त्यांना postfixExp मध्ये जोडा ; जेव्हा तुमचा सामना होतो तेव्हा थांबा
 - a "(" किंवा
 - कमी प्राधान्याचा ऑपरेटर, किंवा
 - जेव्हा स्टॅक रिकामा असतो
 - नंतर ऑपरेटरला स्टॅकवर ढकलून द्या
 - ")" infixExp स्ट्रिंगवर,
 - पॉप ऑपरेटर स्टॅक बंद आणि
 - postfixExp च्या शेवटी जोडा
 - जोपर्यंत तुम्हाला जुळणारे "(" भेटत नाही तोपर्यंत
 - infixExp च्या शेवटी , स्टॅकची उर्वरित सामग्री postfixExp मध्ये जोडा

इन्फिक्स एक्सप्रेशनचे मूल्यांकन

पोस्टफिक्स एक्सप्रेशनचे मूल्यांकन करण्यासाठी अल्गोरिदम

स्टेप 1: पोस्टफिक्स एक्सप्रेशनच्या शेवटी एक ")" जोडा

स्टेप 2: पोस्टफिक्स एक्सप्रेशनचे प्रत्येक वर्ण स्कॅन करा आणि ")" येईपर्यंत चरण 3 आणि 4 पुन्हा करा

स्टेप 3:

ऑपरेंड आढळल्यास , ते स्टॅकवर पुश करा

जर ऑपरेटर X समोर आला, तर

- स्टॅकमधील शीर्ष दोन घटक A आणि B म्हणून पॉप करा
 - BXA चे मूल्यमापन करा, जिथे A हा सर्वात वरचा घटक होता आणि B हा A च्या खाली घटक होता.
 - स्टॅकवर मूल्यांकनाचा परिणाम पुश करा
- [ज.र समाप्त]

स्टेप 4: स्टॅकच्या सर्वात वरच्या घटकाप्रमाणे निकाल सेट करा

स्टेप 5: बाहेर पडा

आता या अल्गोरिदमचा वापर करणारे उदाहरण घेऊ.

$(3 * 4) + 8 / 4$ म्हणून दिलेल्या इन्फिक्स एक्सप्रेसशनचा विचार करा .

पोस्टफिक्स नोटेशन वापरून $9 - ((3 * 4) + 8) / 4$ "9 3 4 * 8 + 4 / -" असे लिहिले जाऊ शकते.

इन्फिक्स एक्सप्रेसशन विचारात घ्या: $(A - B / C) * (A / K - L)$

- स्टेप 1: इन्फिक्स स्ट्रिंग उलट करा. स्ट्रिंग उलट करताना लक्षात घ्या तुम्ही डाव्या आणि उजव्या कंसाची अदलाबदल केली पाहिजे.
 $(L - K / A) * (C / B - A)$
- स्टेप 2: इन्फिक्सची संबंधित पोस्टफिक्स एक्सप्रेसशन मिळवा
 स्टेप 1 च्या परिणामी प्राप्त एक्सप्रेसशन .
 एक्सप्रेसशन आहे: $(L - K / A) * (C / B - A)$
 म्हणून, $[L - (KA /)] * [(CB /) - A]$
 $= [LKA / -] * [CB / A -]$
 $= LKA / - CB / A - *$
- स्टेप 3: उपसर्ग एक्सप्रेसशन मिळविण्यासाठी पोस्टफिक्स एक्सप्रेसशन उलट करा
 म्हणून, उपसर्ग एक्सप्रेसशन $* - A / BC - / AKL$ आहे

इन्फिक्स नोटेशन प्रीफिक्स एक्सप्रेसशनमध्ये रूपांतरित करणे

(Program to convert an Infix expression to Prefix form.)

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>
#define MAX 50
struct infix
{
    char target[MAX];
    char stack[MAX];
    char *s, *t;
    int top, l;
};
void initinfix ( struct infix * );
void setexpr ( struct infix *, char * );
void push ( struct infix *, char );
char pop ( struct infix * );
void convert ( struct infix * );
```

```

int priority ( char c ) ;
void show ( struct infix ) ;
void main( )
{
    struct infix q ;
    char expr[MAX] ;
    clrscr ( ) ;
    initinfix ( &q ) ;
    printf ( "\nEnter an expression in infix form: " ) ;
    gets ( expr ) ;
    setexpr ( &q, expr ) ;
    convert ( &q ) ;
    printf ( "The Prefix expression is: " ) ;
    show ( q ) ;
    getch ( ) ;
}
/* initializes elements of structure variable */
void initinfix ( struct infix *pq )
{
    pq -> top = -1 ;
    strcpy ( pq -> target, "" ) ;
    strcpy ( pq -> stack, "" ) ;
    pq -> l = 0 ;
}
/* reverses the given expression */
void setexpr ( struct infix *pq, char *str )
{
    pq -> s = str ;
    strrev ( pq -> s ) ;
    pq -> l = strlen ( pq -> s ) ;
    *( pq -> target + pq -> l ) = '\0' ;
    pq -> t = pq -> target + ( pq -> l - 1 ) ;
}
/* adds operator to the stack */
void push ( struct infix *pq, char c )
{
    if ( pq -> top == MAX - 1 )
        printf ( "\nStack is full.\n" ) ;
    else
    {
        pq -> top++ ;
        pq -> stack[pq -> top] = c ;
    }
}
/* pops an operator from the stack */

```

```

char pop ( struct infix *pq )
{
    if ( pq -> top == -1 )
    {
        printf ( "Stack is empty\n" ) ;
        return -1 ;
    }
    else
    {
        char item = pq -> stack[pq -> top] ;
        pq -> top-- ;
        return item ;
    }
}

/* converts the infix expr. to prefix form */
void convert ( struct infix *pq )
{
    char opr ;
    while ( *( pq -> s ) )
    {
        if ( *( pq -> s ) == ' ' || *( pq -> s ) == '\t' )
        {
            pq -> s++ ;
            continue ;
        }
        if ( isdigit ( *( pq -> s ) ) || isalpha ( *( pq -> s ) ) )
        {
            while ( isdigit ( *( pq -> s ) ) || isalpha ( *( pq -> s ) ) )
            {
                *( pq -> t ) = *( pq -> s ) ;
                pq -> s++ ;
                pq -> t-- ;
            }
        }
        if ( *( pq -> s ) == ')')
        {
            push ( pq, *( pq -> s ) ) ;
            pq -> s++ ;
        }
        if ( *( pq -> s ) == '*' || *( pq -> s ) == '+' || *( pq -> s ) == '/' ||
            *( pq -> s ) == '%' || *( pq -> s ) == '-' || *( pq -> s ) == '$' )
        {
            if ( pq -> top != -1 )
            {
                opr = pop ( pq ) ;

```

```

        while ( priority ( opr ) > priority ( *( pq -> s ) ) )
        {
            *( pq -> t ) = opr ;
            pq -> t-- ;
            opr = pop ( pq ) ;
        }
        push ( pq, opr ) ;
        push ( pq, *( pq -> s ) ) ;
    }
    else
        push ( pq, *( pq -> s ) ) ;
    pq -> s++ ;
}
if ( *( pq -> s ) == '(' )
{
    opr = pop ( pq ) ;
    while ( opr != ')' )
    {
        *( pq -> t ) = opr ;
        pq -> t-- ;
        opr = pop ( pq ) ;
    }
    pq -> s++ ;
}
}
while ( pq -> top != -1 )
{
    opr = pop ( pq ) ;
    *( pq -> t ) = opr ;
    pq -> t-- ;
}
pq -> t++ ;
}
/* returns the priority of the operator */
int priority ( char c )
{
    if ( c == '$' )
        return 3 ;
    if ( c == '*' || c == '/' || c == '%' )
        return 2 ;
    else
    {
        if ( c == '+' || c == '-' )
            return 1 ;
    }
}

```



```

        else
            return 0 ;
    }
}
/* displays the prefix form of given expr. */
void show ( struct infix pq )
{
    while ( *( pq.t ) )
    {
        printf ( " %c", *( pq.t ) ) ;
        pq.t++ ;
    }
}

```

• इन्फिक्स एक्सप्रेशन पोस्टफिक्स एक्सप्रेशनमध्ये रूपांतरित करणे

(Program to convert an Infix form to Postfix form)

```

#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>

#define MAX 50
struct infix
{
    char target[MAX] ;
    char stack[MAX] ;
    char *s, *t ;
    int top ;
} ;
void initinfix ( struct infix * ) ;
void setexpr ( struct infix *, char * ) ;
void push ( struct infix *, char ) ;
char pop ( struct infix * ) ;
void convert ( struct infix * ) ;
int priority ( char ) ;
void show ( struct infix ) ;
void main( )
{
    struct infix p ;
    char expr[MAX] ;
    initinfix ( &p ) ;
    clrscr( ) ;
    printf ( "\nEnter an expression in infix form: " ) ;
    gets ( expr ) ;
    setexpr ( &p, expr ) ;
    convert ( &p ) ;
}

```

```

    printf ( "\nThe postfix expression is: " );
    show ( p );
    getch( ) ;
}
/* initializes structure elements */
void initinfix ( struct infix *p )
{
    p -> top = -1 ;
    strcpy ( p -> target, "" ) ;
    strcpy ( p -> stack, "" ) ;
    p -> t = p -> target ;
    p -> s = "" ;
}
/* sets s to point to given expr. */
void setexpr ( struct infix *p, char *str )
{
    p -> s = str ;
}
/* adds an operator to the stack */
void push ( struct infix *p, char c )
{
    if ( p -> top == MAX )
        printf ( "\nStack is full.\n" ) ;
    else
    {
        p -> top++ ;
        p -> stack[p -> top] = c ;
    }
}
/* pops an operator from the stack */
char pop ( struct infix *p )
{
    if ( p -> top == -1 )
    {
        printf ( "\nStack is empty.\n" ) ;
        return -1 ;
    }
    else
    {
        char item = p -> stack[p -> top] ;
        p -> top-- ;
        return item ;
    }
}
/* converts the given expr. from infix to postfix form */

```

```

void convert ( struct infix *p )
{
    char opr ;
    while ( *( p -> s ) )
    {
        if ( *( p -> s ) == ' ' || *( p -> s ) == '\t' )
        {
            p -> s++ ;
            continue ;
        }
        if ( isdigit ( *( p -> s ) ) || isalpha ( *( p -> s ) ) )
        {
            while ( isdigit ( *( p -> s ) ) || isalpha ( *( p -> s ) ) )
            {
                *( p -> t ) = *( p -> s ) ;
                p -> s++ ;
                p -> t++ ;
            }
        }
        if ( *( p -> s ) == '(' )
        {
            push ( p, *( p -> s ) ) ;
            p -> s++ ;
        }
        if ( *( p -> s ) == '*' || *( p -> s ) == '+' || *( p -> s ) == '/' || *( p -> s ) == '%' || *( p ->
s ) == '-' || *( p -> s ) == '$' )
        {
            if ( p -> top != -1 )
            {
                opr = pop ( p ) ;
                while ( priority ( opr ) >= priority ( *( p -> s ) ) )
                {
                    *( p -> t ) = opr ;
                    p -> t++ ;
                    opr = pop ( p ) ;
                }
                push ( p, opr ) ;
                push ( p, *( p -> s ) ) ;
            }
            else
                push ( p, *( p -> s ) ) ;
            p -> s++ ;
        }
        if ( *( p -> s ) == ')' )
        {

```

```

        opr = pop ( p ) ;
        while ( ( opr ) != '(' )
        {
            *( p -> t ) = opr ;
            p -> t++ ;
            opr = pop ( p ) ;
        }
        p -> s++ ;
    }
}
while ( p -> top != -1 )
{
    char opr = pop ( p ) ;
    *( p -> t ) = opr ;
    p -> t++ ;
}
*( p -> t ) = '\0' ;
}
/* returns the priority of an operator */
int priority ( char c )
{
    if ( c == '$' )
        return 3 ;
    if ( c == '*' || c == '/' || c == '%' )
        return 2 ;
    else
    {
        if ( c == '+' || c == '-' )
            return 1 ;
        else
            return 0 ;
    }
}
}
/* displays the postfix form of given expr. */
void show ( struct infix p )
{
    printf ( " %s", p.target ) ;
}
}

```

- पोस्टफिक्स एक्सप्रेसशन प्रीफिक्स एक्सप्रेसशनमध्ये रूपांतरित करणे
(Program to convert expression in postfix form to prefix form)

```

#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 50
struct postfix

```

```

{
    char stack[MAX][MAX], target[MAX] ;
    char temp1[2], temp2[2] ;
    char str1[MAX], str2[MAX], str3[MAX] ;
    int i, top ;
} ;
void initpostfix ( struct postfix * ) ;
void setexpr ( struct postfix *, char * ) ;
void push ( struct postfix *, char * ) ;
void pop ( struct postfix *, char * ) ;
void convert ( struct postfix * ) ;
void show ( struct postfix ) ;
void main()
{
    struct postfix q ;
    char expr[MAX] ;
    clrscr() ;
    initpostfix ( &q ) ;
    printf ( "\nEnter an expression in postfix form: " ) ;
    gets ( expr ) ;
    setexpr ( &q, expr ) ;
    convert ( &q ) ;
    printf ( "\nThe Prefix expression is: " ) ;
    show ( q ) ;
    getch() ;
}
/* initializes the elements of the structure */
void initpostfix ( struct postfix *p )
{
    p -> i = 0 ;
    p -> top = -1 ;
    strcpy ( p -> target, "" ) ;
}
/* copies given expr. to target string */
void setexpr ( struct postfix *p, char *c )
{
    strcpy ( p -> target, c ) ;
}
/* adds an operator to the stack */
void push ( struct postfix *p, char *str )
{
    if ( p -> top == MAX - 1 )
        printf ( "\nStack is full." ) ;
    else
        {

```

```

        p -> top++;
        strcpy ( p -> stack[p -> top], str ) ;
    }
}
/* pops an element from the stack */
void pop ( struct postfix *p, char *a )
{
    if ( p -> top == -1 )
        printf ( "\nStack is empty." ) ;
    else
    {
        strcpy ( a, p -> stack[p -> top] ) ;
        p -> top-- ;
    }
}
/* converts given expr. to prefix form */
void convert ( struct postfix *p )
{
    while ( p -> target[p -> i] != '\0' )
    {
        /* skip whitespace, if any */
        if ( p -> target[p -> i] == ' ' )
            p -> i++ ;
        if( p -> target[p -> i] == '%' || p -> target[p -> i] == '*' ||
            p -> target[p -> i] == '-' || p -> target[p -> i] == '+' ||
            p -> target[p -> i] == '/' || p -> target[p -> i] == '$' )
        {
            pop ( p, p -> str2 ) ;
            pop ( p, p -> str3 ) ;
            p -> temp1[0] = p -> target[ p -> i] ;
            p -> temp1[1] = '\0' ;
            strcpy ( p -> str1, p -> temp1 ) ;
            strcat ( p -> str1, p -> str3 ) ;
            strcat ( p -> str1, p -> str2 ) ;
            push ( p, p -> str1 ) ;
        }
        else
        {
            p -> temp1[0] = p -> target[p -> i] ;
            p -> temp1[1] = '\0' ;
            strcpy ( p -> temp2, p -> temp1 ) ;
            push ( p, p -> temp2 ) ;
        }
        p -> i++ ;
    }
}

```

```

    }
    /* displays the prefix form of expr. */
    void show ( struct postfix p )
    {
        char *temp = p.stack[0] ;
        while ( *temp )
        {
            printf ( "%c ", *temp ) ;
            temp++ ;
        }
    }
}

```

• पोस्टफिक्स एक्सप्रेशन इन्फिक्स एक्सप्रेशनमध्ये रूपांतरित करणे

(Program to convert an expression in postfix form to an infix form)

```

#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 50
struct postfix
{
    char stack[MAX][MAX], target[MAX] ;
    char temp1[2], temp2[2] ;
    char str1[MAX], str2[MAX], str3[MAX] ;
    int i, top ;
} ;
void initpostfix ( struct postfix * ) ;
void setexpr ( struct postfix *, char * ) ;
void push ( struct postfix *, char * ) ;
void pop ( struct postfix *, char * ) ;
void convert ( struct postfix * ) ;
void show ( struct postfix ) ;
void main()
{
    struct postfix q ;
    char expr[MAX] ;
    clrscr ( ) ;
    initpostfix ( &q ) ;
    printf ( "\nEnter an expression in postfix form: " ) ;
    gets ( expr ) ;
    setexpr ( &q, expr ) ;
    convert ( &q ) ;
    printf ( "\nThe infix expression is: " ) ;
    show ( q ) ;
    getch ( ) ;
}

```

```

/* initializes data member */
void initpostfix ( struct postfix *p )
{
    p -> i = 0 ;
    p -> top = -1 ;
    strcpy ( p -> target, "" ) ;
}
/* copies given expression to target string */
void setexpr ( struct postfix *p, char *c )
{
    strcpy ( p -> target, c ) ;
}
/* adds an expr. to the stack */
void push ( struct postfix *p, char *str )
{
    if ( p -> top == MAX - 1 )
        printf ( "\nStack is full." ) ;
    else
    {
        p -> top++ ;
        strcpy ( p -> stack[p -> top], str ) ;
    }
}
/* pops an expr. from the stack */
void pop ( struct postfix *p, char *a )
{
    if ( p -> top == -1 )
        printf ( "\nStack is empty." ) ;
    else
    {
        strcpy ( a, p -> stack[p -> top] ) ;
        p -> top-- ;
    }
}
/* converts given expr. to infix form */
void convert ( struct postfix *p )
{
    while ( p -> target[p -> i] )
    {
        /* skip whitespace, if any */
        if( p -> target[p -> i] == ' ' )
            p -> i++ ;
        if ( p -> target[p -> i] == '%' || p -> target[p -> i] == '*' ||
            p -> target[p -> i] == '-' || p -> target[p -> i] == '+' ||
            p -> target[p -> i] == '/' || p -> target[p -> i] == '$' )

```



```

        {
            pop ( p, p -> str2 );
            pop ( p, p -> str3 );
            p -> temp1[0] = p -> target[p -> i] ;
            p -> temp1[1] = '\0' ;
            strcpy ( p -> str1, p -> str3 );
            strcat ( p -> str1, p -> temp1 );
            strcat ( p -> str1, p -> str2 );
            push ( p, p -> str1 );
        }
    else
    {
        p -> temp1[0] = p -> target[p -> i] ;
        p -> temp1[1] = '\0' ;
        strcpy ( p -> temp2, p -> temp1 );
        push ( p, p -> temp2 );
    }
    p -> i++;
}
}
/* displays the expression */
void show ( struct postfix p )
{
    char *t ;
    t = p.stack[0] ;
    while ( *t )
    {
        printf ( "%c ", *t ) ;
        t++ ;
    }
}

```

4.4 रिकर्षण (Recursion):

- रिकर्षण हा स्टॅक एडीटीचा अंतर्निहित अनुप्रयोग आहे.
- रिकर्सिव्ह फंक्शन असे फंक्शन आहे जे स्वतःला कॉल करण्याची आवश्यकता नसलेला अंतिम कॉल होईपर्यंत त्याच्या टास्कच्या छोट्या आवृत्तीचे निराकरण करण्यासाठी स्वतःला कॉल करते.
- प्रत्येक रिकर्सिव्ह सोल्यूशनमध्ये दोन प्रमुख केसेस असतात: बेस केस ज्यामध्ये समस्या समान फंक्शनला आणखी कॉल न करता थेट सोडवता येण्याइतकी सोपी असते.
- आवर्ती केस, ज्यामध्ये प्रथम समस्या सोप्या उपभागांमध्ये विभागली जाते. दुसरे फंक्शन स्वतः कॉल करते परंतु पहिल्या चरणात प्राप्त झालेल्या समस्येच्या उपभागांसह. तिसरा, परिणाम आहे
- सोप्या उप-भागांचे समाधान एकत्र करून प्राप्त केले.

रिकर्षणचे प्रकार (Types of Resursion)

कोणतेही रिकर्षण कार्य यावर आधारित वैशिष्ट्यीकृत केले जाऊ शकते :

1. फंक्शन स्वतःला प्रत्यक्ष किंवा अप्रत्यक्षपणे कॉल करते की नाही (डायरेक्ट किंवा इन्डायरेक्ट रिकर्षण).

2. प्रत्येक रिकर्सिव्ह कॉलवर कोणतेही ऑपरेशन प्रलंबित आहे की नाही (टेल(tail) रिकर्षण किंवा नाही).
3. कॉलिंग पॅटर्नची रचना (लिनिअर किंवा ट्री-रिकर्सिव्ह).

a. डायरेक्ट रिकर्षण (Direct Recursion)

एखादे फंक्शन स्पष्टपणे स्वतःला कॉल करत असल्यास ते **डायरेक्ट** रिकर्सिव्ह असल्याचे म्हटले जाते .
उदाहरणार्थ, खाली दिलेल्या फंक्शनचा विचार करा.

```
int Func( int n)
{
    if(n==0)
        return n;
    return (Func(n-1));
}
```

b. इन्डायरेक्ट रिकर्षण (Indirect Recursion)

एखाद्या फंक्शनला **इन्डायरेक्ट रिकर्षण** होणार असे म्हटले जाते जर त्यात दुसऱ्या फंक्शनला कॉल असेल जे शेवटी कॉल करते. खाली दिलेली फंक्शन्स पहा. ही दोन कार्ये अप्रत्यक्षपणे रिकर्षण होणारी आहेत कारण ते दोघे एकमेकांना कॉल करतात.

```
int Func1(int n)
{
    if(n==0)
        return n;
    return Func2(n);
}
int Func2(int x)
{
    return Func1(x-1);
}
```

c. लिनिअर रिकर्षण (Linear Recursion)

रिकर्सिव्ह फंक्शन्स देखील ज्या पद्धतीने रिकर्षण वाढतात त्यानुसार वैशिष्ट्यीकृत केले जाऊ शकते : लिनिअर पद्धतीने रचना तयार करणे. सोप्या शब्दात, रिकर्सिव्ह फंक्शन लिनिअर रीरिकर्सिव्ह असे म्हटले जाते जेव्हा कोणत्याही प्रलंबित ऑपरेशनमध्ये फंक्शनला दुसरा रिकर्सिव्ह कॉल समाविष्ट नसतो.

उदाहरणार्थ, फॅक्टोरियल फंक्शन लिनिअररीत्या आवर्ती आहे कारण प्रलंबित ऑपरेशनमध्ये फक्त गुणाकार करणे समाविष्ट आहे आणि दुसऱ्या कॉल टू फॅक्ट() फंक्शनचा समावेश नाही.

d. ट्री रिकर्षण (Tree Recursion)

प्रलंबित ऑपरेशनने फंक्शनला दुसरा रिकर्सिव्ह कॉल केल्यास रीरिकर्सिव्ह फंक्शन ट्री रिकर्सिव्ह (किंवा नॉन-लिनीयरली रिकर्सिव्ह) असल्याचे म्हटले जाते .

• उदाहरणार्थ, Fibonacci फंक्शन Fib ज्यामध्ये प्रलंबित ऑपरेशन्स रिकर्सिव्हली Fib फंक्शनला कॉल करतात.

```
int Fibonacci(int num)
{
    if(num <= 2)
        return 1;
    return ( Fibonacci (num - 1) + Fibonacci(num - 2));
}
```

e. टेल रिकर्षण (Tail Recursion)

त्याच्या कॉलरकडे परत आल्यावर कोणतीही ऑपरेशन्स करण्यासाठी प्रलंबित नसल्यास टेल रिकर्सिव्ह असे म्हटले जाते. म्हणजे, जेव्हा कॉल केलेले फंक्शन परत येते, तेव्हा परत केलेले मूल्य कॉलिंग फंक्शनमधून त्वरित परत केले जाते. टेल रिकर्सिव्ह फंक्शन्स अत्यंत इष्ट आहेत कारण ते वापरण्यासाठी अधिक कार्यक्षम आहेत कारण त्यांच्या बाबतीत, सिस्टम स्टॅकवर किती माहिती संग्रहित करावी लागेल ती रिकर्सिव्ह कॉलच्या संख्येपेक्षा स्वतंत्र आहे.

```
int Fact(n)
{
    return Fact1(n, 1);
}
int Fact1(int n, int res)
{
    if (n==1)
        return res;
    return Fact1(n-1, n*res);
}
```

साधक(PROS):

1. न येणाऱ्या उपायांपेक्षा लहान आणि सोपे असतात .
2. कोड अधिक स्पष्ट आणि वापरण्यास सोपा आहे.
3. समस्यांचे निराकरण करण्यासाठी विभाजन करा आणि विजय मिळवा.
4. काही (मर्यादित) घटनांमध्ये, रिकर्षण अधिक कार्यक्षम असू शकते.

बाधक(CONS):

1. काही प्रोग्रामर आणि वाचकांसाठी, रिकर्षण ही एक कठीण संकल्पना आहे.
2. सिस्टम स्टॅक वापरून रिकर्षण लागू केली जाते. प्रणालीवरील स्टॅक जागा मर्यादित असल्यास, सखोल स्तरावर रिकर्षण लागू करणे कठीण होईल.
3. मध्यप्रवाहात पुनरावर्ती प्रक्रिया रद्द करणे धीमे आहे.
4. रिकर्सिव्ह फंक्शन वापरल्याने त्याच्या नॉन-रिकर्सिव्ह भागाच्या तुलनेत कार्यान्वित होण्यासाठी अधिक मेमरी आणि वेळ लागतो.
5. बग शोधणे कठीण आहे, विशेषतः ग्लोबल व्हेरिएबल्स वापरताना.

संदर्भ (References): -

1. Data Structure Using C : Dr. E. Balagurusamy
2. Data structure and Algorithms : K. Mehlhorn
3. Data structure and Algorithms: Dr. Mizanur Rahman
4. Data Structure with c :Lipschutz Schaum Series.

युनिट - 5

क्यू (Queue)

विषय निष्पत्ती (Course Outcome): अॅर्रे आणि लिंकड लिस्ट इम्प्लिमेंटेशन्स वापरून रांगेवर ऑपरेशन्स करा.

घटक निष्पत्ती (Theory Learning Outcomes):

1. अॅर्रे आणि लिंकड लिस्ट वापरून क्यूचे प्रतिनिधित्व करा.
2. वेगवेगळ्या प्रकारच्या क्यूची वैशिष्ट्ये स्पष्ट करा.
3. क्यूत INSERT आणि DELETE ऑपरेशन्स पार पाडण्यासाठी अल्गोरिदम तयार करा.

5.1 क्यू (Queue)

क्यू ही एक नॉन-प्रिमिटिव्ह रेखीय डेटा रचना आहे जी एका टोकाला घटक घालण्याची आणि दुसऱ्या टोकाला घटक काढण्याची परवानगी देते. ज्या शेवटी घटक काढला जातो त्याला म्हणतात फ्रंट (FRONT), आणि ज्या टोकाला नवीन घटक टाकला जाऊ शकतो त्याला रियर (REAR) म्हणतात. घटक काढणे किंवा समाविष्ट करणे अनुक्रमे सूचीच्या पुढील आणि मागील बाजूसच होऊ शकते. क्यूत जोडलेला पहिला घटक सूचीमधून काढून टाकला जाणारा पहिला घटक आहे. त्यामुळे क्यूला **फर्स्ट-इन-फर्स्ट-आउट (FIFO)** यादी असेही संबोधले जाते. 'क्यू' हे नाव या शब्दाच्या दैनंदिन वापरातून आले आहे. एका रेल्वे आरक्षण बूथचा विचार करा, जिथे आम्हाला आरक्षणाच्या क्यूत जावे लागेल. नवीन ग्राहक मागील टोकापासून क्यूत उभे होते, तर ज्या ग्राहकांना त्यांच्या जागा आरक्षित आहेत ते पुढच्या टोकापासून रांगेत निघून जातात. याचा अर्थ ग्राहक ज्या क्रमाने सेवा केंद्रात येतात त्या क्रमाने सेवा दिली जाते (म्हणजे प्रथम येणाऱ्यास प्रथम सेवा प्रकार). तीच वैशिष्ट्ये आमच्या क्यूत लागू होतात.

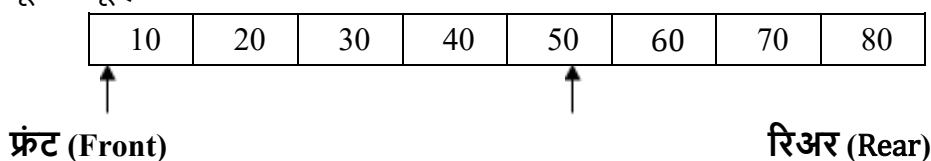


Fig 5.1 : क्यू (Queue)

Fig 5.1 मध्ये, 10 हा पहिला घटक आहे आणि 80 हा रांगेत जोडलेला शेवटचा घटक आहे. त्याचप्रमाणे, 10 हा काढला जाणारा पहिला घटक असेल आणि 80 हा काढला जाणारा शेवटचा घटक असेल. आकृती 5.2(a) ते 5.2(d) अंतर्भूत ऑपरेशन दरम्यान ग्राफिकली क्यू दर्शवते:

F = -1 आणि R = -1

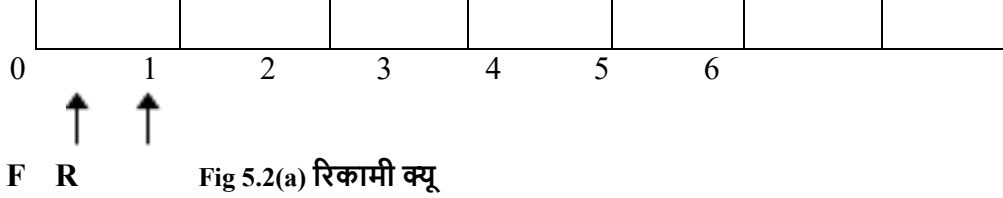


Fig 5.2(a) रिकामी क्यू

F = 0 आणि R = 0

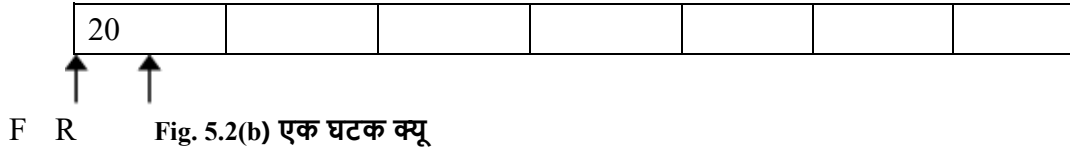


Fig. 5.2(b) एक घटक क्यू

F = 0 आणि R = 1

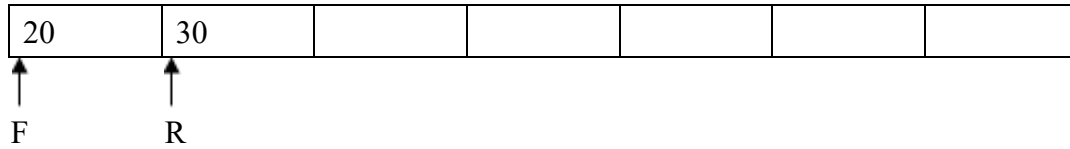


Fig. 5.2(c) दोन घटकांची क्यू

F = 0 आणि R = 2

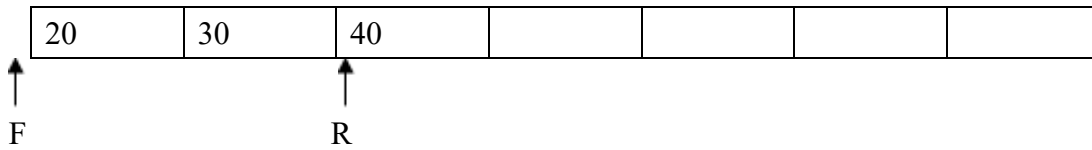


Fig. 5.2(d) दोन घटकांची क्यू

वरील आकड्यांवरून हे स्पष्ट होते की जेव्हाही आपण क्यूत एखादा घटक घालतो, तेव्हा रीअरचे मूल्य एकाने वाढते म्हणजे

$$\text{रीअर} = \text{रीअर} + 1$$

तसेच, रांगेतील पहिला घटक घालताना आम्ही नेहमी समोरचा भाग एकाने वाढवला.

$$\text{फ्रंट} = \text{फ्रंट} + 1$$

संपूर्ण ऑपरेशन दरम्यान मोर्चा बदलला जाणार नाही. हटविण्याच्या ऑपरेशन दरम्यान खालील आकडे ग्राफिकली क्यू दर्शवतात :

F = 1 आणि R = 2

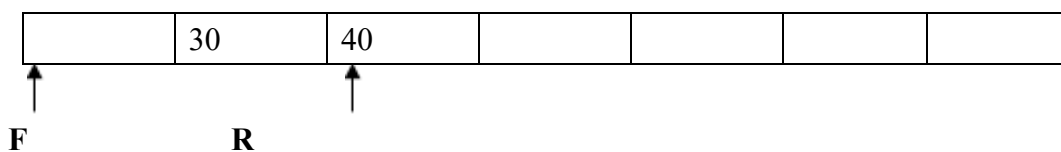


Fig.5.2(e) एक घटक (20) समोरून हटवले

F = 2 आणि R = 2

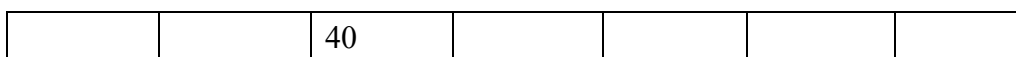




Fig. 5.2(f) दुसरा घटक (30) समोरून हटवला

Fig.5. 2(e) आणि 5.2(f) मधून हे स्पष्ट होते की जेव्हा जेव्हा रांगेतून एखादा घटक काढला जातो तेव्हा फ्रंटचे मूल्य एकाने वाढवले जाते .

$$\text{फ्रंट} = \text{फ्रंट} + 1 \quad (F=F+1)$$

आता, जर आपण रांगेत कोणतेही घटक समाविष्ट केले तर, क्यू अशी दिसेल:

F= 2 आणि R = 3

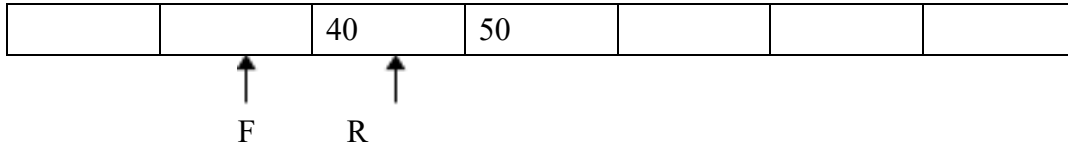


Fig.5.2(g) हटविल्यानंतर समाविष्ट करणे

क्यूची इम्प्लिमेंटेशन (Implementation of Queue)

क्यू दोन प्रकारे लागू केल्या जाऊ शकतात:

1. स्टॅटिक इम्प्लिमेंटेशन (अरे वापरून)
2. डायनॅमिक इम्प्लिमेंटेशन (पॉइंटर्स वापरून)

स्टॅटिक इम्प्लिमेंटेशन (Static implementation):

क्यूची स्टॅटिक इम्प्लिमेंटेशन अरेद्वारे दर्शविली जाते . जर क्यू अरे वापरून अंमलात आणली असेल, तर आम्हाला क्यूत किती घटक संग्रहित करायचे आहेत याची खात्री असणे आवश्यक आहे, कारण आम्हाला डिझाइनच्या वेळी किंवा प्रक्रिया सुरू होण्यापूर्वी अरेचा आकार घोषित करावा लागेल. या प्रकरणात, अरेची सुरुवात क्यूसाठी पुढची असेल आणि अरेचे शेवटचे स्थान क्यूसाठी मागील म्हणून काम करेल. Fig. (5.3) क्यूचे प्रतिनिधित्व अरे म्हणून दाखवते .

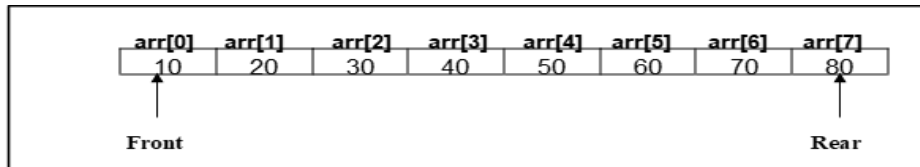


Fig.5.3: अरे म्हणून क्यूचे प्रतिनिधित्व

अरे वापरून अंमलात आणल्यावर खालील संबंध क्यूत उपस्थित घटकांची एकूण संख्या देते :

$$\text{रिअर} - \text{फ्रंट} + 1$$

हे देखील लक्षात घ्या की जर $\text{फ्रंट} > \text{रिअर}$ असेल तर क्यूत कोणताही घटक नसेल किंवा क्यू रिकामी असेल.

क्यू ऑपरेशन्स (Queue operations)

क्यूत करता येणारी मूलभूत ऑपरेशन्स आहेत:

1. क्यूत घटक इन्सर्ट करणे (Insert)
2. क्यूतील घटक काढणे. (Delete)
3. क्यूतील सर्व घटक ट्रॅव्हर्स करणे. (Traverse)

रेखीय रांगेत समाविष्ट करणे आणि हटवणे यासाठी अल्गोरिदम आणि कार्ये (अरे वापरत आहे)

1. लिनिअर क्यूत इन्सर्ट करण्यासाठी अल्गोरिदम

QUEUE[MAXSIZE] हा रेखीय रांगेच्या अंमलबजावणीसाठी एक अर्रे आहे आणि NUM हा घटक रेखीय रांगेत घालायचा आहे, FRONT रांगेच्या सुरुवातीला घटकाचा अनुक्रमणिका क्रमांक दर्शवतो आणि REAR हा घटकाचा अनुक्रमणिका क्रमांक दर्शवतो. रांगेचा शेवट.

पायरी 1 : जर REAR = (MAXSIZE -1): तर

लिहा : "रांग ओव्हरफ्लो" आणि परत या

[If संरचनेचा शेवट]

पायरी 2 : रेखीय रांगेत घालण्यासाठी NUM वाचा.

पायरी 3 : REAR सेट करा := REAR + 1

पायरी 4 : रांग सेट करा [मागील] := NUM

पायरी 5 : जर समोर = -1 : तर समोर = 0 सेट करा.

[If संरचनेचा शेवट]

पायरी 6: बाहेर पडा

लिनिअर क्यूट इन्सर्ट करण्यासाठी कोड (अर्रे वापरून)

```
void qinsert() {
    int num;
    if(rear==MAXSIZE-1) {
        printf("\nQueue is full (Queue overflow)");
        return;
    }
    printf("\nEnter the element to be inserted : ");
    scanf("%d",&num);
    rear++;
    queue[rear]=num;
    if(front==-1) front=0;
}
```

2. लिनिअर क्यूट डिलीट करण्यासाठी अल्गोरिदम

QUEUE[MAXSIZE] हा रेखीय रांग लागू करण्यासाठी एक अर्रे आहे आणि NUM हा घटक रेखीय रांगेतून हटवायचा आहे, FRONT रांगेच्या सुरुवातीला घटकाचा अनुक्रमणिका क्रमांक दर्शवतो आणि REAR हा घटकाचा अनुक्रमणिका क्रमांक दर्शवतो. रांगेचा शेवट.

पायरी 1 : जर समोर = -1 : तर

लिहा: "रांग अंडरफ्लो" आणि परत या

[If संरचनेचा शेवट]

पायरी 2 : सेट NUM := रांग[समोर]

पायरी 3 : "हटवलेला आयटम आहे:" लिहा, NUM

पायरी 4 : समोर सेट करा := समोर + 1.

पायरी 5 : जर समोर > मागील : तर

समोर सेट करा := मागील := -1.

[If संरचनेचा शेवट]

पायरी 6: बाहेर पडा

```
void lqdelete()
{
if(front == -1)
{
printf("\nQueue is empty (Queue underflow)");
return; }
int num;
num=queue[front];
printf("\nDeleted element is : %d",num);
front++;
if(front>rear)
front=rear=-1;
}
```

Program 1 : Static implementation of Linear Queues using arrays

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAXSIZE 5
void initialize();
void lqinsert();
void lqdelete();
void lqtraverse();
int queue[MAXSIZE];
int front,rear;
void main()
{
clrscr();
initialize();
int choice;
while(1)
{
clrscr();
printf("\nSTATIC IMPLEMENTATION OF LINEAR QUEUE");
printf("\n-----");
printf("\n1. Insert");
printf("\n2. Delete");
printf("\n3. Traverse");
printf("\n4. Exit");
printf("\n-----");
printf("\n\nEnter your choice [1/2/3/4] : ");
scanf("%d",&choice);
switch(choice)
{
```



```

case 1 : lqinsert();
    break;
case 2 : lqdelete();
    break;
case 3 : lqtraverse();
    break;
case 4 : exit(0);
default : printf("\nInvalid choice");
}
getch();
}
}
// Function to initialize queue
void initialize()
{
    front=rear=-1;
}
// Function to insert an element into queue
void lqinsert()
{
    int num;
    if(rear==MAXSIZE-1)
    {
        printf("\nQueue is full (Queue overflow)");
        return;
    }
    printf("\nEnter the element to be inserted : ");
    scanf("%d",&num);
    rear++;
    queue[rear]=num;
    if(front==-1)
        front=0;
}
// Function for Delete an element from queue
void lqdelete()
{
    if(front==-1)
    {
        printf("\nQueue is empty (Queue underflow)");
        return;
    }
    int num;
    num=queue[front];
    printf("\nDeleted element is : %d",num);
    front++;

```

```

if(front>rear)
    front=rear=-1;
}
// Function to display Queue
void lqtraverse()
{
    if(front== -1)
    {
        printf("\nQueue is empty (Queue underflow)");
        return;
    }
    else
    {
        printf("\nQueue elements are : \n");
        for(int i=front;i<=rear;i++)
            printf("%d\t",queue[i]);
    }
}

```

2. लिनिअर क्यूचे डायनॅमिक इम्प्लिमेंटेशन

लिनिअर क्यूत समाविष्ट करणे आणि हटवणे यासाठी अल्गोरिदम (वापरणे सूचक). क्यू अशी रचना असू द्या ज्याची घोषणा खालीलप्रमाणे दिसते:

```

struct queue {
    int info;
    struct queue *link;
} *start=NULL;

```

डायनॅमिकसाठी एका लिनिअर क्यूत घालण्यासाठी (Insertion) आणि हटवण्यासाठी (Deletion) अल्गोरिदम लिंकड लिस्ट वापरून इम्प्लिमेंटेशन

a. लिनिअर क्यूत घटक घालण्यासाठी अल्गोरिदम (Insert):

चला PTR हा स्ट्रक्चर पॉइंटर आहे जो नवीन नोडसाठी मेमरी वाटप करतो आणि NUM हा घटक रेखीय क्यूत समाविष्ट केला जाणार आहे, INFO नोडचा माहितीचा भाग दर्शवतो आणि LINK लिंक किंवा पुढील पॉइंटर पुढील नोडच्या ऍड्रेसकडे निर्देशित करतो. FRONT पहिल्या नोडचा ऍड्रेस दर्शवतो, REAR शेवटच्या नोडचा ऍड्रेस दर्शवतो. सुरुवातीला, क्यूत पहिला घटक टाकण्यापूर्वी, FRONT=REAR=NULL.

स्टेप 1: PTR वापरून नवीन नोडसाठी मेमरी वाटप करा.

स्टेप 2 : लिनिअर क्यूत घालण्यासाठी NUM वाचा.

स्टेप 3 : PTR->INFO = NUM सेट करा

स्टेप 4 : PTR->LINK= NULL सेट करा

स्टेप 5 : जर फ्रंट = शून्य : तर

FRONT=REAR=PTR सेट करा

अन्यथा

REAR->LINK=PTR सेट करा;

REAR=PTR सेट करा;

[इफ इल्स स्ट्रक्चरचा शेवट]

स्टेप 6: बाहेर पडा

लिनिअर क्यूत घटक घालण्यासाठी कोड (Code):

```
void lqinsert()
{
    struct queue *ptr;
    int num;
    ptr=(struct queue*)malloc(sizeof(struct queue));
    printf("\nEnter element to be inserted in queue : ");
    scanf("%d",&num);
    ptr->info=num;
    ptr->link=NULL;
    if(front==NULL)
    {
        front=ptr;
        rear=ptr;
    }
    else
    {
        rear->link=ptr;
        rear=ptr;
    }
}
```

a. लिनिअर क्यूत घटक हटवण्यासाठी अल्गोरिदम (Delete):

चला पीटीआर(PTR) हा स्ट्रक्चर पॉइंटर आहे जो रेखीय क्यूतील पहिल्या नोडची मेमरी डिलॉक करतो आणि NUM हा घटक क्यूतून हटवायचा आहे, INFO हटवलेल्या नोडचा माहितीचा भाग दर्शवतो आणि LINK हटवलेल्या नोड पॉइंटिंगची लिंक किंवा पुढील पॉइंटर दर्शवते. पुढील नोडच्या ऍड्रेसवर. FRONT पहिल्या नोडचा ऍड्रेस दर्शवतो, REAR शेवटच्या नोडचा ऍड्रेस दर्शवतो.

स्टेप 1 : जर फ्रंट = शून्य : तर

'Queue is Empty(Queue Underflow)' लिहा आणि परत या.

[If संरचनेचा शेवट]

स्टेप 2 : PTR = FRONT सेट करा

स्टेप 3 : NUM = PTR->INFO सेट करा

स्टेप 4 : 'रेखीय रांगेतून हटवलेले घटक is : ',NUM लिहा.

स्टेप 5 : FRONT = FRONT->LINK

स्टेप 6 : FRONT = NULL असल्यास : नंतर REAR = NULL सेट करा.

[इफ स्ट्रक्चरचा शेवट].

स्टेप 7 : पीटीआर वापरून रांगेच्या सुरुवातीला नोडची मेमरी डिलॉकेट करा.

स्टेप 8: बाहेर पडा.

//Function(Procedure) for deleting a node from a Linear Queue

```
void qdelete()
```

```
{
```

```

if(front==NULL)
{
printf("\nQueue is empty (Queue underflow)");
return;
}
struct queue *ptr;
int num;
ptr=front;
num=ptr->info;
printf("\nThe deleted element is : %d",num);
front=front->link;
if(front==NULL)
rear=NULL;
free(ptr);
}
}

```

Program 2 : Dynamic implementation of linear queue using pointers

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct queue
{
int info;
struct queue *link;
}*front,*rear;
void initialize();
void qinsert();
void qdelete();
void qtraverse();
void main()
{
int choice;
initialize();
while(1)
{
clrscr();
printf("\nDYNAMIC IMPLEMENTATION OF LINEAR QUEUE");
printf("\n1. Insert");
printf("\n2. Delete");
printf("\n3. Traverse");
printf("\n4. Exit");
printf("\n\nEnter your choice [1/2/3/4] : ");
scanf("%d",&choice);

```

```

switch(choice)
{
case 1: qinsert();
        break;
case 2: qdelete();
        break;
case 3: qtraverse();
        break;
case 4: exit(0);;
default : printf("\nInvalid choice");
}
getch();
}
}
// Function for initialize linear Queue
void initialize()
{
    front=rear=NULL;
}
// Function to insert element in Linear queue
void qinsert() {
    struct queue *ptr;
    int num;
    ptr=(struct queue*)malloc(sizeof(struct queue));
    printf("\nEnter element to be inserted in queue : ");
    scanf("%d",&num);
    ptr->info=num;
    ptr->link=NULL;
    if(front==NULL) {
        front=ptr;
        rear=ptr;
    }
    else
    {
        rear->link=ptr;
        rear=ptr;
    }
}
// Function to delete element from Linear queue
void qdelete() {
    if(front==NULL) {
        printf("\nQueue is empty (Queue underflow)");
        return;
    }
    struct queue *ptr;

```

```

int num;
ptr=front;
num=ptr->info;
printf("\nThe deleted element is : %d",num);
front=front->link;
if(front==NULL)
    rear=NULL;
free(ptr);
}
// Function to display Linear Queue
void qtraverse()
{
    struct queue *ptr;
    if(front==NULL) {
        printf("\nQueue is empty (Queue underflow)");
        return;
    }
    else {
        ptr=front;
        printf("\n\nQueue elements are : \n");
        printf("\nROOT");
        while(ptr!=NULL) {
            printf(" -> %d",ptr->info);
            ptr=ptr->link;
        }
        printf(" -> NULL");
    }
}

```

b. सर्क्युलर क्यू (Circular Queue)

आम्ही अरे वापरून लागू केलेली क्यू एका मर्यादित ग्रेड आहे. त्या इम्प्लीमेंटेशनमध्ये अशी शक्यता आहे की क्यू पूर्ण म्हणून नोंदवली गेली आहे (रिअर बाजू अरेच्या शेवटी पोहोचली आहे), जरी प्रत्यक्षात रांगेच्या सुरुवातीला रिक्त स्लॉट असू शकतात. या मर्यादित मात करण्यासाठी आपण क्यू सर्क्युलर क्यू म्हणून लागू करू शकतो. येथे जसे आपण क्यूत घटक जोडत जातो आणि अरेच्या शेवटी पोहोचतो तेव्हा पुढील घटक अरेच्या पहिल्या स्लॉटमध्ये संग्रहित केला जातो (जर तो विनामूल्य असेल). समजा अरे **अरे** n घटकांचा वापर वर्तुळाकार क्यू लागू करण्यासाठी केला जातो ज्यावर आपण पोहोचू शकतो **arr [n -1]**.

आम्ही अरेच्या शेवटी पोहोचलो असल्याने क्यूत आणखी घटक जोडू शकत नाही. क्यू पूर्ण झाल्याचा अहवाल देण्याऐवजी, जर क्यूतील काही घटक हटवले गेले असतील तर क्यूच्या सुरुवातीला रिक्त स्लॉट असू शकतात. अशा परिस्थितीत हे स्लॉट क्यूत नवीन घटक जोडून भरले जातील. थोडक्यात आम्ही अरेच्या शेवटी पोहोचलो असल्यामुळे, क्यू पूर्ण झाली म्हणून नोंदवली जाणार नाही. जेव्हा अरे स्टॅंडमधील सर्व स्लॉट व्यापलेले असतील तेव्हाच क्यू पूर्ण झाल्याचा अहवाल दिला जाईल. Fig.5.4 सर्क्युलर क्यूचे सचित्र प्रतिनिधित्व दाखवते.

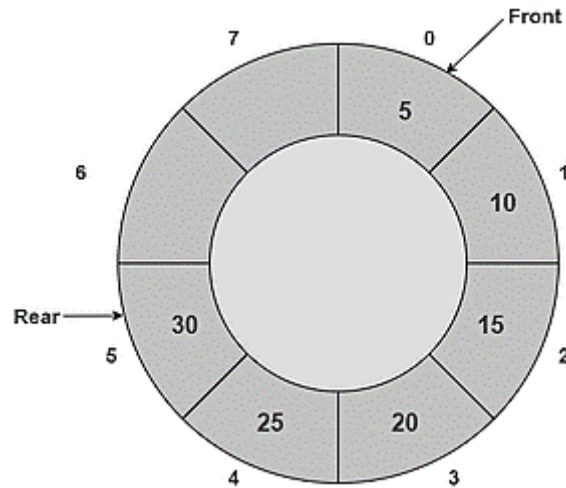


Fig.5.4: सर्क्युलर क्यूचे सचित्र प्रतिनिधित्व

सर्क्युलर क्यूत घालण्यासाठी (Insert) आणि हटवण्यासाठी (Delete) अल्गोरिदम (अॅरे वापरून)

1. सर्क्युलर क्यूत घालण्यासाठी (Insert) अल्गोरिदम (अॅरे वापरून)

CQUEUE [MAXSIZE] हे परिपत्रक रांग लागू करण्यासाठी एक अॅरे आहे, जेथे MAXSIZE कमाल दर्शवते. अॅरेचा आकार . NUM हा घटक गोलाकार रांगेत घालायचा आहे, FRONT रांगेच्या सुरुवातीला घटकाचा अनुक्रमणिका क्रमांक दर्शवतो आणि REAR रांगेच्या शेवटी घटकाचा अनुक्रमणिका क्रमांक दर्शवतो.

पायरी 1 : जर समोर = (मागील + 1) % MAXSIZE :

तर लिहा : "रांग ओव्हरफ्लो" आणि परत या. [If संरचनेचा शेवट]

पायरी 2 : परिपत्रक रांगेत घालण्यासाठी NUM वाचा.

पायरी 3 : जर FRONT = -1 : तर FRONT = REAR = 0 सेट करा.

बाकी

REAR = (REAR + 1) % MAXSIZE सेट करा.

[अन्यतर रचनाचा शेवट]

पायरी 4 : सेट CQUEUE[REAR]=NUM;

पायरी 5: बाहेर पडा

Function(Procedure) to Delete an element from a Queue

```
void cqdelete()
{
    int num;
    if(front==-1)
    {
        printf("\nQueue is Empty (Queue underflow)");
        return;
    }
    num=cqueue[front];
```

```

printf("\nDeleted element from circular queue is : %d",num);
if(front==rear)
front=rear=-1;
else
front=(front+1)%MAXSIZE;
}

```

Program 3 : Static implementation of Circular queue using arrays

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAXSIZE 5
void cqinsert();
void cqdelete();
void cqdisplay();
int cqueue[MAXSIZE];
int front=-1,rear=-1;
void main()
{
int choice;
while(1)
{
clrscr();
printf("\nSTATIC IMPLEMENTATION OF CIRCULAR QUEUE");
printf("\n-----");
printf("\n1. Insert");
printf("\n2. Delete");
printf("\n3. Traverse");
printf("\n4. Exit");
printf("\n-----");
printf("\n\nEnter your choice [1/2/3/4] : ");
scanf("%d",&choice);
switch(choice)
{
case 1 : cqinsert();
break;
case 2 : cqdelete();
break;
case 3 : cqdisplay();
break;
case 4 : exit(0);
default : printf("\nInvalid choice");
}
getch();
} }
// Function to insert element in the Circular Queue

```



```

void cqinsert()
{
    int num;
    if(front==(rear+1)%MAXSIZE)
    {
        printf("\nQueue is Full(Queue overflow)");
        return;
    }
    printf("\nEnter the element to be inserted in circular queue : ");
    scanf("%d",&num);
    if(front==-1)
        front=rear=0;
    else
        rear=(rear+1) % MAXSIZE;
    cqueue[rear]=num;
}
// Function to delete element from the circular queue
void cqdelete()
{
    int num;
    if(front==-1)
    {
        printf("\nQueue is Empty (Queue underflow)");
        return;
    }
    num=cqueue[front];
    printf("\nDeleted element from circular queue is : %d",num);
    if(front==rear)
        front=rear=-1;
    else
        front=(front+1)%MAXSIZE;
}
// Function to display circular queue
void cqdisplay()
{
    int i;
    if(front==-1)
    {
        printf("\nQueue is Empty (Queue underflow)");
        return;
    }
    printf("\n\nCircular Queue elements are : \n");
    for(i=front;i<=rear;i++)
        printf("\ncqueue[%d] : %d",i,cqueue[i]);
    if(front>rear)

```

```

{
for(i=0;i<=rear;i++)
printf("cqueue[%d] : %d\n",i,cqueue[i]);
for(i=front;i<MAXSIZE;i++)
printf("cqueue[%d] : %d\n",i,cqueue[i]);
}
}

```

लिनिअर क्यूवर सक्च्युलर क्यूचे फायदे :

कमाल सह लिनिअर क्यूत. आकार 5, अरेच्या शेवटच्या स्थानावर (4) घटक घालल्यानंतर, घटक समाविष्ट केले जाऊ शकत नाहीत, कारण क्यूत नवीन घटक नेहमी रिअर टोकापासून घातले जातात आणि येथे रिअर बाजू अरेच्या शेवटच्या स्थानास सूचित करते (सबस्क्रिप्टसह स्थान 4) फ्रंटच्या आधी सुरू होणारी स्थाने विनामूल्य असली तरीही. पण सक्च्युलर क्यूत, क्यूच्या शेवटच्या ठिकाणी घटक असल्यास, आपण अरेच्या सुरुवातीला एक नवीन घटक घालू शकतो.

प्रायोरिटी क्यू (Priority queue)

प्रायोरिटी क्यू हा घटकांचा संग्रह आहे जेथे घटक त्यांच्या प्रायोरिटी स्तरांनुसार संग्रहित केले जातात. घटक ज्या क्रमाने जोडले किंवा काढले जातात ते घटकाच्या प्राधान्याने ठरवले जातात.

प्रायोरिटी क्यू राखण्यासाठी खालील नियम लागू केले जातात :

- (1) उच्च प्रायोरिटी असलेल्या घटकावर कमी प्राधान्याच्या कोणत्याही घटकापूर्वी प्रक्रिया केली जाते .
- (2) घटक असल्यास , क्यूत प्रथम जोडलेल्या घटकावर प्रक्रिया केली जाईल.

ऑपरेटिंग सिस्टीम मध्ये जॉब शेड्यूलिंग लागू करण्यासाठी प्रायोरिटी क्यूचा वापर केला जातो जेथे उच्च प्रायोरिटी असलेल्या प्रोसेसवर प्रथम प्रक्रिया केली जाते. अग्रक्रम क्यूचा आणखी एक अनुप्रयोग म्हणजे सिम्युलेशन सिस्टीम जिथे प्रायोरिटी इव्हेंटच्या वेळेशी संबंधित असते.

मेमरीमध्ये प्रायोरिटी क्यू राखण्याचे मुख्यतः दोन मार्ग आहेत. एक वापरते सिंगल लिस्ट, आणि दुसरी सिंगल क्यू वापरते. प्रायोरिटी क्यू त घटक जोडण्यात किंवा हटवण्यात सहज किंवा अवघडपणा स्पष्टपणे एखाद्याने निवडलेल्या प्रतिनिधित्वावर अवलंबून असते.

प्रायोरिटी क्यूचे सिंगल लिस्ट प्रतिनिधित्व (Representation of Priority Queue using Single List):

मेमरीमध्ये अग्रक्रम क्यू राखण्याचा एक मार्ग म्हणजे सिंगल लिस्ट, खालीलप्रमाणे :

- (a) सूचीतील प्रत्येक नोडमध्ये माहितीच्या तीन बाबी असतील; माहिती फील्ड INFO, प्रायोरिटी क्रमांक PRN आणि लिंक क्रमांक LINK.
- (b) सूचीतील नोड Y च्या आधी एक नोड X आहे
 - (i) जेव्हा X ला उच्च प्रायोरिटी असते तेव्हा Y आणि
 - (ii) जेव्हा दोघांना समान प्रायोरिटी असते परंतु Y च्या आधी X ला सूचीमध्ये जोडले जाते . याचा अर्थ असा की सिंगल लिस्टतील क्रम प्रायोरिटी क्यू च्या क्रमांशी संबंधित आहे.

प्रायोरिटीक्रम क्यू नेहमीच्या पद्धतीने काम करतील : प्रायोरिटी क्रमांक जितका कमी तितका प्रायोरिटी जास्त.

प्रायोरिटी क्यू चे अरे प्रतिनिधित्व(Representation of Priority Queue using array):

मेमरीमध्ये अग्रक्रम क्यू राखण्याचा दुसरा मार्ग म्हणजे प्रत्येक स्तरासाठी (किंवा प्रत्येक प्रायोरिटी क्रमांकासाठी) स्वतंत्र क्यू वापरणे. अशी प्रत्येक क्यू त्याच्या स्वतःच्या सक्च्युलर अरेमध्ये दिसेल आणि त्याच्या स्वतःच्या पॉइंटर्सची जोडी, FRONT आणि REAR असणे आवश्यक आहे. खरं तर, प्रत्येक क्यूला समान प्रमाणात जागा दिली जाते ,

लिनिअर अँरेऐवजी द्विमितीय अँरे QUEUE वापरली जाऊ शकते. प्रायोरिटी क्यू चे प्रतिनिधित्व करण्याच्या या दोन मार्गांपैकी, प्रायोरिटी क्यूचे अँरे प्रतिनिधित्व सिंगल लिस्टपेक्षा अधिक वेळ-कार्यक्षम आहे. याचे कारण असे की सिंगल लिस्टमध्ये घटक जोडताना, सूचीवर एक लिनिअर शोध करणे आवश्यक आहे. दुसरीकडे, अग्रक्रम क्यूचे सिंगल लिस्ट प्रतिनिधित्व अँरे प्रतिनिधित्वापेक्षा अधिक जागा-कार्यक्षम असू शकते. कारण अँरे वापरताना ओव्हरफ्लो होतो जेव्हा कोणत्याही एका प्रायोरिटी स्तरावरील घटकांची संख्या त्या पातळीच्या क्षमतेपेक्षा जास्त असते, परंतु सिंगल लिस्ट वापरताना, घटकांची एकूण संख्या एकूण क्षमतेपेक्षा जास्त असते तेव्हाच ओव्हरफ्लो होतो. दुसरा पर्याय म्हणजे प्रत्येक प्रायोरिटी स्तरासाठी लिंक केलेली लिस्ट वापरणे.

अँप्लिकेशन्स (Applications):

1. प्रोसेसर शेड्युलिंगसाठी राउंड रॉबिन तंत्र क्यू वापरून लागू केले जाते .
2. सर्व प्रकारच्या ग्राहक सेवा (जसे की रेल्वे तिकीट आरक्षण) सेंटर सॉफ्टवेअर्स ग्राहकांची माहिती साठवण्यासाठी क्यूचा वापर करून डिझाइन केलेले आहेत
3. प्रिंटर सर्व्हर रूटीन क्यू वापरून डिझाइन केले आहेत. अनेक वापरकर्ते प्रिंटर सर्व्हरचा वापर करून प्रिंटर शेअर करतात (एक समर्पित संगणक ज्यावर प्रिंटर जोडलेला असतो), प्रिंटर सर्व्हर नंतर सर्व वापरकर्त्यांकडील सर्व जॉब्स एका क्यूत सर्व्हरच्या हार्ड डिस्कवर स्पूल करतो. येथून क्यूतील त्यांच्या संख्येनुसार नोकरी एक-एक करून छापली जाते .

Reference book: -

1. Data Structure Using C : Dr. E. Balagurusamy
2. Data structure and Algorithms : K. Mehlhorn
3. Data structure and Algorithms: Dr. Mizanur Rahman
4. Data Structure with c :Lipschutz Schaum Series.

युनिट – 6

ट्री (Tree)

विषय निष्पत्ती (Course Outcome): समस्यांचे निराकरण करण्यासाठी ट्री तयार करा आणि मार्गक्रमण (ट्रॅव्हर्स) करा.

घटक निष्पत्ती (Theory Learning Outcomes):

1. दिलेल्या ट्री टर्मिनॉलॉजीचे वर्णन करा.
2. प्रदान केलेल्या डेटावर आधारित बायनरी सर्च (Binary Search) ट्री तयार करा.
3. दिलेल्या पद्धतीचा वापर करून ट्री ट्रॅव्हर्स (Traverse) करण्यासाठी अल्गोरिदम तयार करा.
4. एक एक्स्प्रेसन ट्री (Expression Tree) तयार करा.
5. हिप(Heap) तयार करा

6.1 ट्री टर्मिनॉलॉजीचा परिचय (Introduction to Tree Terminologies)

ट्री डेटा स्ट्रक्चर एक महत्वाची संकल्पना आहे जी संगणक विज्ञानात डेटा व्यवस्थापनासाठी वापरली जाते. हे एक हायरेरिकल स्वरूप आहे जिथे डेटा नोड्सच्या रूपात साठवला जातो आणि त्यांच्यात संबंध असतात. यात मध्यवर्ती नोड, स्ट्रक्चरल नोड्स आणि सब-नोड्स असतात, जे एजद्वारे जोडलेले असतात. आम्ही असेही म्हणू शकतो की ट्री च्या डेटा स्ट्रक्चरमध्ये, रूट एजेस आणि लीफ नोडजोडलेली आहेत.

ट्री डेटा स्ट्रक्चर ही एक श्रेणीबद्ध रचना आहे जी नेव्हिगेट करणे आणि शोधणे सोपे आहे अशा प्रकारे डेटाचे प्रतिनिधित्व आणि आयोजन करण्यासाठी वापरली जाते. हा नोड्सचा संग्रह आहे जो कडांनी जोडलेला आहे आणि नोड्समध्ये श्रेणीबद्ध संबंध आहे.

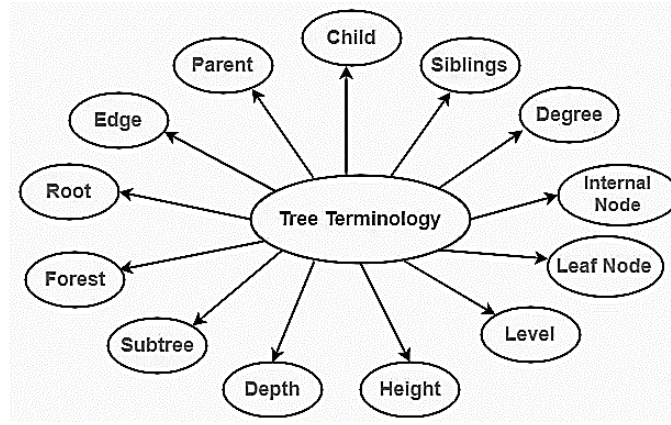


Figure 6.2: ट्री टर्मिनॉलॉजी (Tree Terminology)

6.1.1 ट्री चे प्रकार (Types of Tree) :-

1) सर्वसाधारण ट्री (General tree):-

सामान्य ट्री ही अक्रमित ट्री डेटा संरचना आहे जिथे रूट नोडमध्ये कमीतकमी 0 किंवा जास्तीत जास्त 'एन' उपट्री असतात. सामान्य ट्रीना त्यांच्या पदानुक्रमावर कोणतेही बंधन घातलेले नाही. रूट नोड अशा प्रकारे इतर सर्व उपवृक्षांच्या (Sub Tree) सुपरसेटप्रमाणे कार्य करते.

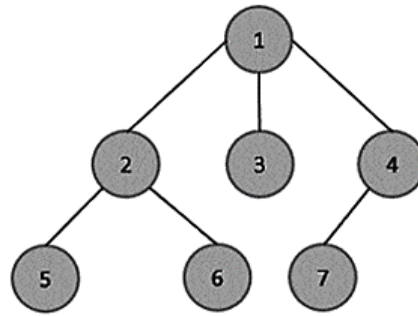


Fig. 6.3: सर्वसाधारण ट्री (General Tree)

2) बायनरी ट्री (Binary tree):-

बायनरी ट्री ही एक ट्री डेटा स्ट्रक्चर (नॉन-लिनियर) आहे ज्यामध्ये प्रत्येक नोडमध्ये जास्तीत जास्त दोन चाईल्ड असू शकतात ज्यांना डावे चाईल्ड आणि उजवे चाईल्ड म्हणून संबोधले जाते. बायनरी ट्रीतील सर्वात वरच्या नोडला रूट नोड म्हणतात आणि सर्वात खालच्या नोडसला लीफ नोड म्हणतात. बायनरी ट्रीला वरच्या बाजूला रूट नोड आणि खालच्या बाजूला लीफ नोड असलेली श्रेणीबद्ध रचना म्हणून कल्पना केली जाऊ शकते.

बायनरी ट्रीचे प्रतिनिधित्व (Representation of Binary tree):-

बायनरी ट्रीमधील प्रत्येक नोडचे तीन भाग असतात:

1. डेटा (data)
2. डाव्या रूट पॉइंटर (लेफ्ट चाईल्डचा ऍड्रेस)
3. उजवा रूट पॉइंटर (राईट चाईल्डचा ऍड्रेस)

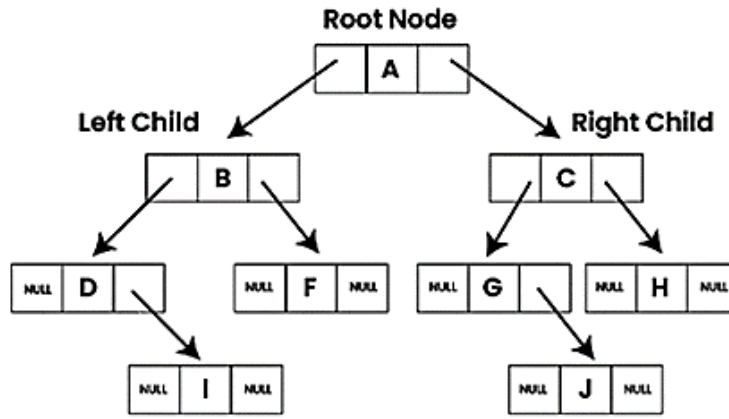


Fig. 6.4: (बायनरी ट्री) Binary tree

3. बायनरी सर्च ट्री (Binary search Tree-BST) :-

बायनरी सर्च ट्री (बीएसटी) हा एक विशेष प्रकारचा बायनरी ट्री आहे ज्यामध्ये नोडच्या डाव्या (नोड लेफ्ट चाइल्ड नोड) चे मूल्य रूट नोडच्या मूल्यापेक्षा कमी असते आणि उजव्या नोड (राईट चाइल्ड नोड) चे मूल्य रूट नोडच्या मूल्यापेक्षा जास्त असते. या गुणधर्माला बीएसटी गुणधर्म म्हणतात आणि यामुळे ट्री तील घटक कार्यक्षमतेने शोधणे, घालणे आणि हटविणे शक्य होते.

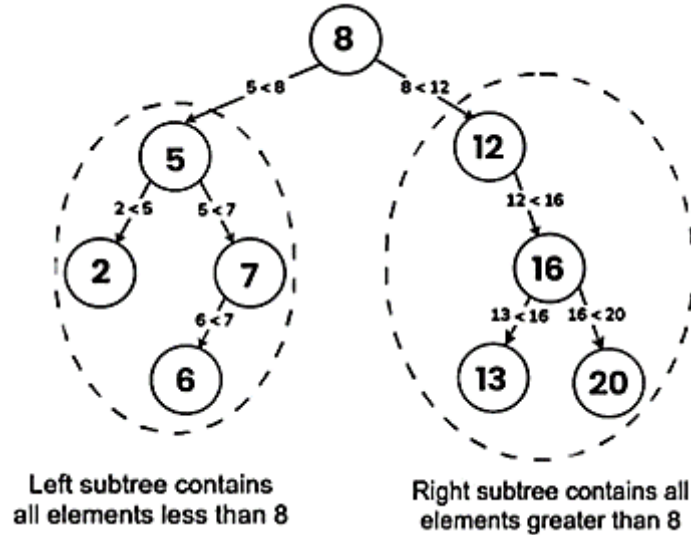


Fig. 6.5: बायनरी सर्च ट्री (Binary Search tree)

बायनरी सर्च ट्रीचे गुणधर्म (Features of Binary Search Tree):-

- नोडच्या डाव्या सबट्रीमध्ये नोडच्या कीपेक्षा कमी मूल्य असलेल्या नोड्स असतात.
- नोडच्या उजव्या सबट्रीमध्ये नोडच्या कीपेक्षा मोठ्या मूल्य असलेल्या नोड्सच असतात.
- याचा अर्थ मुळाच्या डावीकडील प्रत्येक गोष्ट मुळाच्या मूल्यापेक्षा कमी असते आणि मुळाच्या उजवीकडील सर्व काही मुळाच्या मूल्यापेक्षा जास्त असते. या कामगिरीमुळे, बायनरी शोध खूप सोपा आहे.
- डावा आणि उजवा उपट्री प्रत्येकी बायनरी सर्च ट्री असावा.
- डुप्लिकेट नोड्स नसावेत (बीएसटीमध्ये वेगवेगळ्या हाताळणी पध्दतींसह डुप्लिकेट मूल्ये असू शकतात)

C Program to build a binary search tree from arrays.

```

#include <stdio.h>
#include <conio.h>
#include <alloc.h>

```

```

struct node
{
    struct node *left ;
    char data ;
    struct node *right ;
} ;
struct node * buildtree ( int ) ;
void inorder ( struct node * ) ;
char arr[ ] = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', '\0', '\0', 'H' } ;
int lc[ ] = { 1, 3, 5, -1, 9, -1, -1, -1, -1, -1 } ;
int rc[ ] = { 2, 4, 6, -1, -1, -1, -1, -1, -1, -1 } ;
void main( )
{
    struct node *root ;
    clrscr( ) ;
    root = buildtree ( 0 ) ;
    printf ( "In-order Traversal:\n" ) ;
    inorder ( root ) ;
    getch( ) ;
}
struct node * buildtree ( int index )
{
    struct node *temp = NULL ;
    if ( index != -1 )
    {
        temp = ( struct node * ) malloc ( sizeof ( struct node ) ) ;
        temp -> left = buildtree ( lc[index] ) ;
        temp -> data = arr[index] ;
        temp -> right = buildtree ( rc[index] ) ;
    }
    return temp ;
}
void inorder ( struct node *root )
{
    if ( root != NULL )
    {
        inorder ( root -> left ) ;
        printf ( "%c\t", root -> data ) ;
        inorder ( root -> right ) ;
    }
}

```

6.2 ट्री ट्रॅव्हर्सल तंत्र (Tree Traversal Techniques):-

ट्री ट्रॅव्हर्सल तंत्रात **ट्री** च्या सर्व नोड्सला भेट देण्याचे विविध मार्ग समाविष्ट आहेत. लिनिअर डेटा स्ट्रक्चर्स (लिस्ट, लिंकड लिस्ट, क्यू, स्टॅक्स इ.) च्या विपरीत ज्यांना ते पार करण्याचा एकच तार्किक मार्ग आहे, **ट्री** वेगवेगळ्या मार्गांनी पार केली जाऊ शकतात.

ट्री ट्रॅव्हर्सल म्हणजे **ट्री**च्या प्रत्येक नोडला ठराविक क्रमाने एकदा भेट देण्याची किंवा प्रवेश करण्याची प्रक्रिया. **ट्री ट्रॅव्हर्सल** अल्गोरिदम आम्हाला झाडाच्या सर्व नोड्सला भेट देण्यास आणि प्रक्रिया करण्यास मदत करतात. **ट्री** ही एक रेखीय डेटा रचना नसल्यामुळे, तेथे एकाधिक नोड्स आहेत ज्यांना आपण विशिष्ट नोडला भेट दिल्यानंतर भेट देऊ शकतो. झाडाच्या नोड्सला कोणत्या क्रमाने भेट द्यायची हे ठरविणारी अनेक **ट्री ट्रॅव्हलिंग** तंत्रे आहेत.

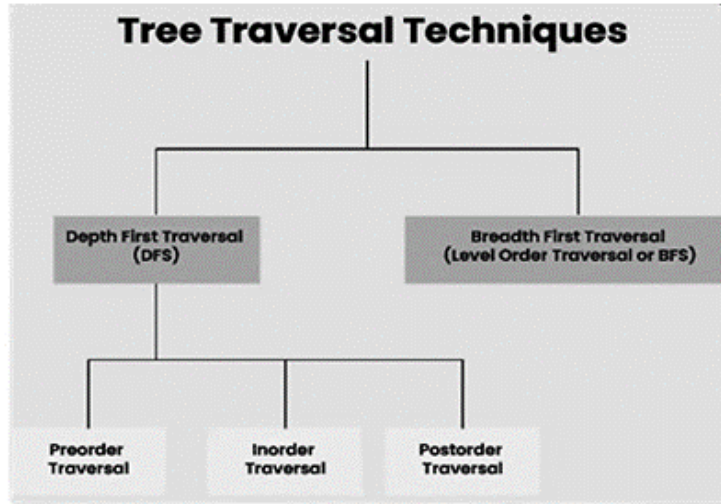
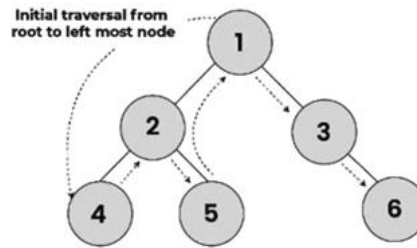


Fig- 6.6 ट्री ट्रॅव्हर्सल तंत्र (Tree Traversal Techniques)

1. इन ऑर्डर ट्रॅव्हर्सल (In-Order Traversal)

क्रमाने नोडला भेट द्या: डावे -> रूट -> उजवे



Inorder Traversal: 4 → 2 → 5 → 1 → 3 → 6

Fig- 6.7: इन ऑर्डर ट्रॅव्हर्सल Inorder Traversal

इनऑर्डर ट्रॅव्हर्सलसाठी अल्गोरिदम (Inorder traversal of Binary tree):-

इनऑर्डर(ट्री)

- डाव्या उपवृक्षाला पार करा, म्हणजे इनऑर्डर (डावे->उपट्री) म्हणा
- मुळाला भेट द्या.
- उजव्या उपवृक्षाला पार करा, म्हणजे इनऑर्डर (उजवा->उपट्री) म्हणा

इनऑर्डर ट्रॅव्हर्सलचे उपयोग (Use of In order traversal):-

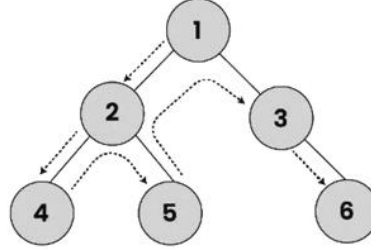
- बायनरी सर्च ट्रीज (बीएसटी) च्या बाबतीत, इनऑर्डर ट्रॅव्हर्सल कमी न होणाऱ्या क्रमाने नोड्स देते.
- वाढत्या क्रमाने बीएसटीचे नोड्स मिळविण्यासाठी, इनऑर्डर ट्रॅव्हलरचा एक प्रकार वापरला जाऊ

शकतो जिथे इनऑर्डर ट्रॅव्हलर उलट आहे.

- एक्सप्रेसन ट्रीमध्ये साठवलेल्या अंकगणितीय एक्सप्रेसनचे मूल्यांकन करण्यासाठी इनऑर्डर ट्रॅव्हर्सल वापर केला जाऊ शकतो.

2. प्रीऑर्डर ट्रॅव्हर्सल (Preorder Traversal):-

प्रीऑर्डर ट्रॅव्हर्सल क्रमाने नोडला भेट देते: **रूट -> डावा -> उजवा**



Preorder Traversal: 1 → 2 → 4 → 5 → 3 → 6

Fig. 6.8: प्रीऑर्डर ट्रॅव्हर्सल (Preorder Traversal)

प्रीऑर्डर ट्रॅव्हर्सलसाठी अल्गोरिदम (Preorder Traversal of Binary tree):-

पूर्वक्रम (ट्री)

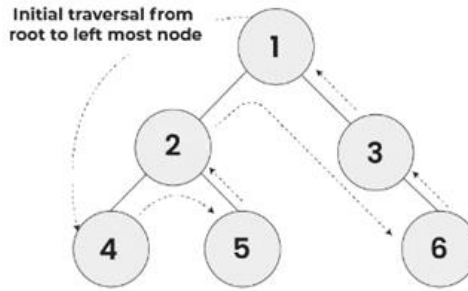
- मुळाला भेट द्या.
- डाव्या उपवृक्षाला पार करा, म्हणजे प्रीऑर्डर (डावे->उपट्री) म्हणा
- उजव्या उपवृक्षाला पार करा, म्हणजेच प्रीऑर्डर (उजवा->सबट्री) म्हणा

प्रीऑर्डर ट्रॅव्हर्सलचे उपयोग (Use of Pre Order Traversal):-

- ट्रीची प्रत तयार करण्यासाठी प्रीऑर्डर ट्रॅव्हर्सलचा वापर केला जातो.
- एक्सप्रेसन वृक्षावर उपसर्ग एक्सप्रेसन मिळविण्यासाठी प्रीऑर्डर ट्रॅव्हलरलदेखील वापरले जाते.

3. पोस्टऑर्डर ट्रॅव्हर्सल (Postorder Traversal) :-

पोस्टऑर्डर ट्रॅव्हल क्रमाने नोडला भेट देते: **डावे -> उजवे -> रूट**



Postorder Traversal: 4 → 5 → 2 → 6 → 3 → 1

Fig. 6.9: पोस्टऑर्डर ट्रॅव्हर्सल Postorder Traversal

पोस्टऑर्डर ट्रॅव्हर्सलसाठी अल्गोरिदम (Postorder Traversal of Binary tree):-

अल्गोरिदम पोस्टऑर्डर (ट्री)

- डाव्या उपवृक्षाला पार करा, म्हणजे पोस्टऑर्डर (डावे->सबट्री) म्हणा
- उजव्या उपवृक्षाला पार करा, म्हणजे पोस्टऑर्डर (उजवे->उपट्री) म्हणा
- मुळाला भेट द्या

पोस्टऑर्डर ट्रॅव्हर्सलचे उपयोग (Use of Post Traversal):-

- एक्सप्रेसन ट्रीची पोस्टफिक्स एक्सप्रेसन मिळविण्यासाठी पोस्टऑर्डर ट्रॅव्हलर देखील उपयुक्त आहे.

- पोस्टऑर्डर ट्रॅव्हर्सल कचरा संकलन अल्गोरिदममध्ये मदत करू शकते, विशेषतः अशा प्रणालींमध्ये जिथे मॅन्युअल मेमरी व्यवस्थापन वापरले जाते.

Program

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct btreenode
{
    struct btreenode *leftchild ;
    int data ;
    struct btreenode *rightchild ;
} ;
void insert ( struct btreenode **, int ) ;
void inorder ( struct btreenode * ) ;
void preorder ( struct btreenode * ) ;
void postorder ( struct btreenode * ) ;
void main( )
{
    struct btreenode *bt ;
    int req, i = 1, num ;
    bt = NULL ; /* empty tree */
    clrscr( ) ;
    printf ( "Specify the number of items to be inserted: " ) ;
    scanf ( "%d", &req ) ;
    while ( i++ <= req )
    {
        printf ( "Enter the data: " ) ;
        scanf ( "%d", &num ) ;
        insert ( &bt, num ) ;
    }
    printf ( "\nIn-order Traversal: " ) ;
    inorder ( bt ) ;
    printf ( "\nPre-order Traversal: " ) ;
    preorder ( bt ) ;
    printf ( "\nPost-order Traversal: " ) ;
    postorder ( bt ) ;
}
/* inserts a new node in a binary search tree */
void insert ( struct btreenode **sr, int num )
{
    if ( *sr == NULL )
    {
        *sr = malloc ( sizeof ( struct btreenode ) ) ;

        ( *sr ) -> leftchild = NULL ;
        ( *sr ) -> data = num ;
        ( *sr ) -> rightchild = NULL ;
    }
}
```

```

        return ;
    }
    else /* search the node to which new node will be attached */
    {
        /* if new data is less, traverse to left */
        if ( num < ( *sr ) -> data )
            insert ( &( ( *sr ) -> leftchild ), num ) ;
        else
            /* else traverse to right */
            insert ( &( ( *sr ) -> rightchild ), num ) ;
    }
    return ;
}
/* traverse a binary search tree in a LDR (Left-Data-Right) fashion */
void inorder ( struct btreenode *sr )
{
    if ( sr != NULL )
    {
        inorder ( sr -> leftchild ) ;

        /* print the data of the node whose leftchild is NULL or the path
        has already been traversed */
        printf ( "\t%d", sr -> data ) ;

        inorder ( sr -> rightchild ) ;
    }
    else
        return ;
}
/* traverse a binary search tree in a DLR (Data-Left-right) fashion */
void preorder ( struct btreenode *sr )
{
    if ( sr != NULL )
    {
        /* print the data of a node */
        printf ( "\t%d", sr -> data ) ;
        /* traverse till leftchild is not NULL */
        preorder ( sr -> leftchild ) ;
        /* traverse till rightchild is not NULL */
        preorder ( sr -> rightchild ) ;
    }
    else
        return ;
}
/* traverse a binary search tree in LRD (Left-Right-Data) fashion */
void postorder ( struct btreenode *sr )
{
    if ( sr != NULL )

```

```

{
    postorder ( sr -> leftchild ) ;
    postorder ( sr -> rightchild ) ;
    printf ( "\t%d", sr -> data ) ;
}
else
    return ;
}

```

6.3 एक्सप्रेशन ट्री(Expression tree) :-

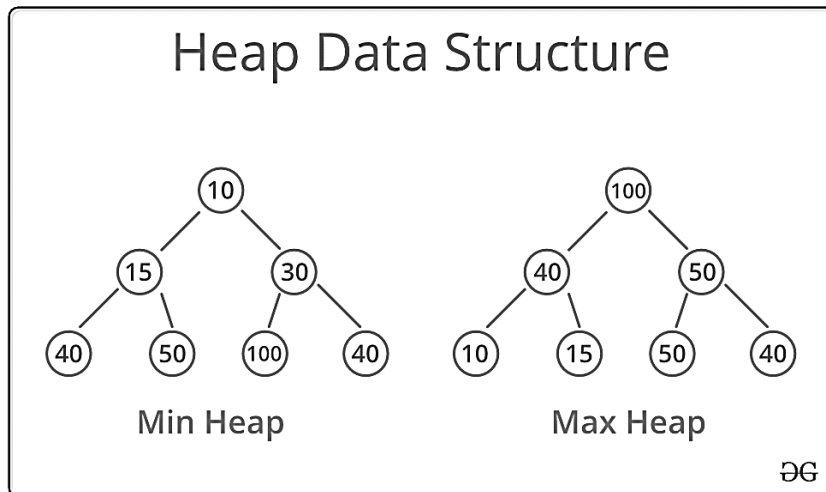
एक्सप्रेशन ट्री हे बायनरी ट्री आहेत जे इनफिक्स, पोस्टफिक्स आणि उपसर्ग एक्सप्रेशन सारख्या विविध एक्सप्रेशन व्यक्त करण्यासाठी वापरले जातात. या एक्सप्रेशन वृक्षाच्या अंतर्गत नोड्समध्ये +, -, *,/,^, इत्यादी ऑपरेटर असतात. आणि एक्सप्रेशन ट्रीच्या बाह्य नोड्समध्ये (लीफ नोड्स) ऑपरेंड असतात, ज्यावर ऑपरेशन केले जातात. एक्सप्रेशन ट्रीच्या पानांमध्ये नेहमीच वर्ण, पूर्णांक, दुहेरी इत्यादी कोणत्याही डेटा प्रकारचे ऑपरेंड असतात.

एक्सप्रेशन ट्रीने दर्शविलेल्या एक्सप्रेशनचे मूल्यमापन :-

एक्सप्रेशन ट्रीद्वारे दर्शविल्या जाणार्या एक्सप्रेशनच्या मूल्यांकनासाठी, आम्ही रिकर्षण वापरतो. एक्सप्रेशनचे मूल्य शोधण्यासाठी आम्ही प्रीऑर्डर ट्रॅव्हलचे अनुसरण करतो. प्रथम डाव्या उपवृक्षावर प्रक्रिया केली जाते आणि नंतर मूळ नोडच्या उजव्या उपवृक्षावर रिकर्षणमध्ये प्रक्रिया केली जाते. एक्सप्रेशन ट्रीपासून एक्सप्रेशनचे मूल्यांकन करण्यासाठी अनुसरण करावयाच्या चरण खाली आहेत:

6.4 हीप डेटा स्ट्रक्चर (Heap data structure)

हीप ही एक संपूर्ण बायनरी ट्री डेटा स्ट्रक्चर आहे जी हीप प्रॉपर्टीचे समाधान करते: प्रत्येक नोडसाठी, त्याच्या मुलांचे मूल्य त्याच्या स्वतःच्या मूल्यापेक्षा मोठे किंवा समान असते. ढीग सहसा प्राधान्य रांग लागू करण्यासाठी वापरले जातात, जेथे सर्वात लहान (किंवा सर्वात मोठा) घटक नेहमी झाडाच्या मुळाशी असतो.



Program:-

```

#include <stdio.h>
#include <conio.h>
void restoreup ( int, int * ) ;
void restoredown ( int, int *, int ) ;
void makeheap ( int *, int ) ;
void add ( int, int *, int * ) ;

```

```

int replace ( int, int *, int ) ;
int del ( int *, int * ) ;
void main()
{
    int arr [20] = { 1000, 7, 10, 25, 17, 23, 27, 16, 19, 37, 42, 4, 33, 1, 5, 11 } ;
    int i, n = 15 ;
    clrscr() ;
    makeheap ( arr, n ) ;
    printf ( "Heap:\n" ) ;
    for ( i = 1 ; i <= n ; i++ )
        printf ( "%d\t", arr [i] ) ;
    i = 24 ;
    add ( i, arr, &n ) ;
    printf ( "\n\nElement added %d.\n", i ) ;
    printf ( "\nHeap after addition of an element:\n" ) ;
    for ( i = 1 ; i <= n ; i++ )
        printf ( "%d\t", arr [i] ) ;
    i = replace ( 2, arr, n ) ;
    printf ( "\n\nElement replaced %d.\n", i ) ;
    printf ( "\nHeap after replacement of an element:\n" ) ;
    for ( i = 1 ; i <= n ; i++ )
        printf ( "%d\t", arr [i] ) ;
    i = del ( arr, &n ) ;
    printf ( "\n\nElement deleted %d.\n", i ) ;
    printf ( "\nHeap after deletion of an element:\n" ) ;
    for ( i = 1 ; i <= n ; i++ )
        printf ( "%d\t", arr [i] ) ;
    getch() ;
}
void restoreup ( int i, int *arr )
{
    int val ;
    val = arr [i] ;
    while ( arr [i / 2] <= val )
    {
        arr [i] = arr [i / 2] ;
        i = i / 2 ;
    }
    arr [i] = val ;
}
void restoredown ( int pos, int *arr, int n )
{

```

```

    int i, val ;
    val = arr [pos] ;
    while ( pos <= n / 2 )
    {
        i = 2 * pos ;
        if ( ( i < n ) && ( arr [i] < arr [i + 1] ) )
            i++ ;
        if ( val >= arr [i] )
            break ;
        arr [pos] = arr [i] ;
        pos = i ;
    }
    arr [pos] = val ;
}
void makeheap ( int *arr, int n )
{
    int i ;
    for ( i = n / 2 ; i >= 1 ; i-- )
        restoredown ( i, arr, n ) ;
}
void add ( int val, int *arr, int *n )
{
    ( *n ) ++ ;
    arr [*n] = val ;
    restoreup ( *n, arr ) ;
}
int replace ( int i, int *arr, int n )
{
    int r = arr [1] ;
    arr [1] = i ;
    for ( i = n / 2 ; i >= 1 ; i-- )
        restoredown ( i, arr, n ) ;
    return r ;
}
int del ( int *arr, int *n )
{
    int val ;
    val = arr [1] ;
    arr [1] = arr [*n] ;
    ( *n ) -- ;
    restoredown ( 1, arr, *n ) ;
    return val ;
}

```

}

Reference book: -

1. Data Structure Using C : Dr. E. Balagurusamy
2. Data structure and Algorithms : K. Mehlhorn
3. Data structure and Algorithms: Dr. Mizanur Rahman
4. Data Structure with c :Lipschutz Schaum Series.