Abstract Data Types (ADTs) – List ADT

Data structures

A data structure is a mathematical or logical way of organizing data in the memory that consider not only the items stored but also the relationship to each other and also it is characterized by accessing functions.

Applications of data structures

- Compiler design
- Operating System
- Database Management system
- Network analysis

Abstract Data Types (ADTs)

Abstract Data type is defined by the set of operations that may be performed on it and by mathematical constraints on the effects of those operations

Abstract Data type is an extension of modular design.

- An abstract data type is a set of operations such as Union, Intersection, Complement, Find etc.,
- The basic idea of implementing ADT is that the operations are written once in program and can be called by any part of the program.

List ADT

List ADT is a sequential storage structure.

- General list of the form a1, a2, a3….., an and the size of the list is 'n'.
- Any element in the list at the position I is defined to be ai, ai+1 the successor of ai and ai-1 is the predecessor of ai.
- Empty list of size 0.

Various operations performed on List

1. Insert (X, 5) - Insert the element X after the position 5.
2. Delete (X) - The element X is deleted
3. Find (X) - Returns the position of X.
4. Next (i) - Returns the position of its successor element i+1.
5. Previous (i) - Returns the position of its predecessor i-1.
6. Print list - Contents of the list is displayed.

Make empty - Makes the list empty

Array-based implementation

Implementation of List ADT

1. Array Implementation
2. Linked List Implementation

1. Array Implementation of List

Array is a collection of specific number of data stored in a consecutive memory location.

- Insertion and Deletion operation are expensive as it requires more data movement
- Find and Printlist operations take constant time.
- Even if the array is dynamically allocated, an estimate of the maximum size of the list is required which considerably wastes the memory space.

List ADT: Array Insertion

Suppose we insert x at position 0. insert(x, 0)

• Must move every element in the array down one position. That takes N operations (one for moving each element in the array).

1, 4, 32, 6, 29, -3 x, 1, 4, 32, 6, 29, -3

• So inserting one number is O(N).

• And inserting N numbers could take up to N2 time!

» We often have to do many inserts – three, twelve, or N.

» That‟s the worst case, depending on where they are inserted

List ADT: Array Deletion

Same problem as insertion.

• On average will have to move at least half the list up one position.

• If don't move elements, then findKth won't work.

• And if delete 0th position will have to move the whole list; N operations or O(N).

• So deleting a single element will be O(N).

• And deleting N elements will take O(N2).

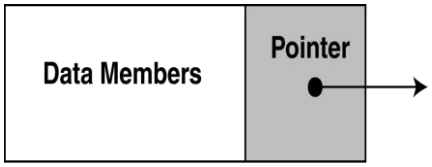
• And we usually have to delete more than once – five, twelve, N times.

2. Linked List Implementation

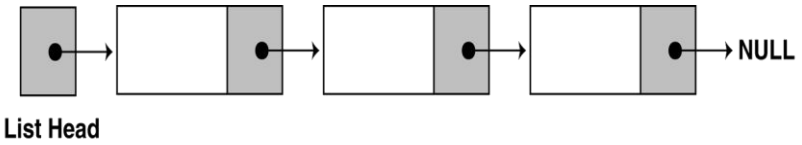
- A linked list is a series of connected *nodes*, where each node is a data structure.
- A linked list can grow or shrink in size as the program runs

The composition of a Linked List

- Each node in a linked list contains one or more members that represent data.
- In addition to the data, each node contains a pointer, which can point to another node.



• A linked list is called "linked" because each node in the series has a pointer that points to the next node in the list.

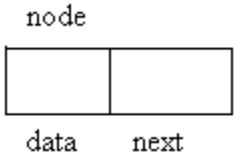


3.1 ***Singly Linked List***

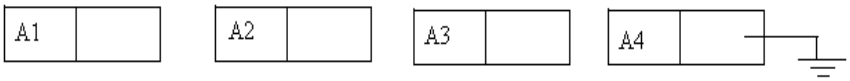
Linked list consists of series of nodes. Each node contains the element and a pointer to its successor node. The pointer of the last node points to NULL.

Insertion and deletion operations are easily performed using linked list.

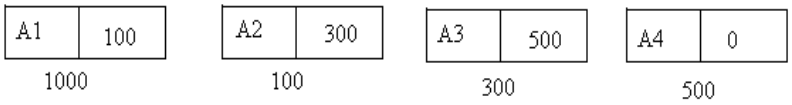
A single node is represented as follows



Actual representation of the singly linked list.



Singly Linked list with actual pointers



Singly linked lists

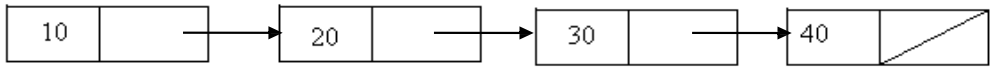
A singly linked list is a linked list in which each node contains only one link field pointing to the next node in the list.

```
#include <iostream>
using namespace std;

// Node class
class Node {
   int data;
   Node* next;

  public:
```

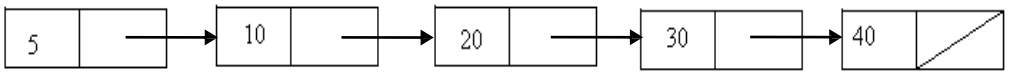
```
    Node() {};
    void SetData(int aData) { data = aData; };
    void SetNext(Node* aNext) { next = aNext; };
    int Data() { return data; };
    Node* Next() { return next; };
};
```

Example: Insert the element '5' in the beginning of the list temp.

Before insertion

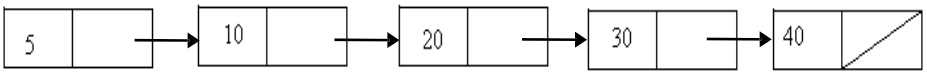


After Insertion :

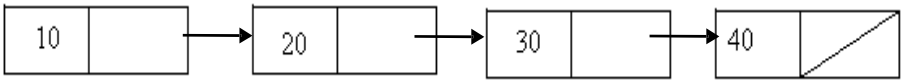



```
void List::Append(int data) {

    // Create a new node
    Node* newNode = new Node();
    newNode->SetData(data);
    newNode->SetNext(NULL);

    // Create a temp pointer
    Node *tmp = head;

    if ( tmp != NULL ) {
    // Nodes already present in the list
    // Parse to end of list
    while ( tmp->Next() != NULL ) {
       tmp = tmp->Next();
    }

    // Point the last node to the new node
    tmp->SetNext(newNode);
    }
    else {
    // First node in the list
    head = newNode;
    }
}
```

DELETION AT FIRST
Before deletion



After deletion




```
/**
 * Delete a node from the list
 */
void List::Delete(int data) {

   // Create a temp pointer
   Node *tmp = head;

   // No nodes
   if ( tmp == NULL )
   return;

   // Last node of the list
   if ( tmp->Next() == NULL ) {
   delete tmp;
   head = NULL;
   }
   else {
   // Parse thru the nodes
   Node *prev;
   do {
      if ( tmp->Data() == data ) break;
      prev = tmp;
      tmp = tmp->Next();
   } while ( tmp != NULL );

   // Adjust the pointers
   prev->SetNext(tmp->Next());

  // Delete the current node
   delete tmp;
   }
}
```

Polynomial Manipulation

Polynomial ADT

A set of values and a set of allowable operations on those values.

Here are the most common operations on a polynomial:

- Add & Subtract
- Multiply
- Differentiate

There are different ways of implementing the polynomial ADT:

1. Array (not recommended)
2. Linked List (preferred and recommended)

1. Array Implementation:

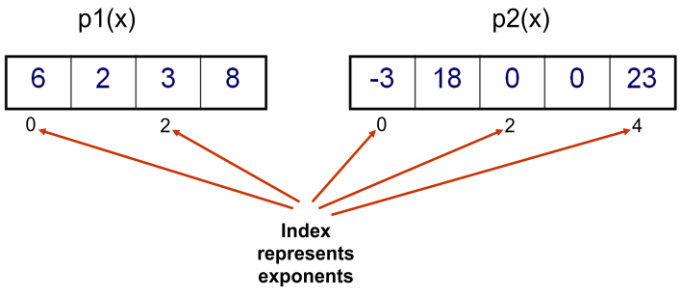
- $p1(x) = 8x^3 + 3x^2 + 2x + 6$
- $p2(x) = 23x^4 + 18x - 3$

Advantages of using an Array:

- Only good for non-sparse polynomials.
- Ease of storage and retrieval.

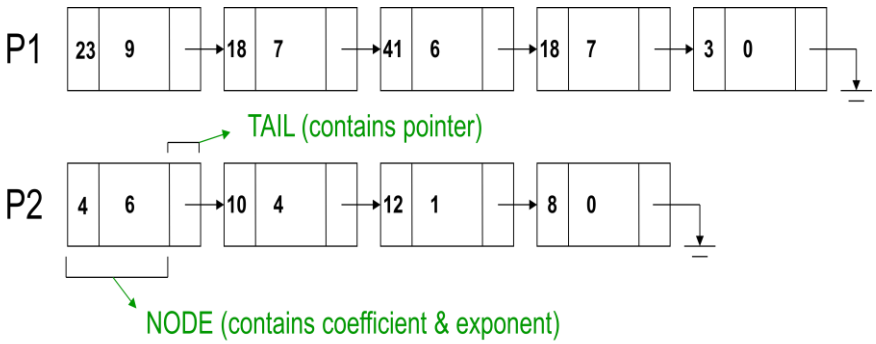
Disadvantages of using an Array:

- Have to allocate array size ahead of time.
- Huge array size required for sparse polynomials. Waste of space and runtime.



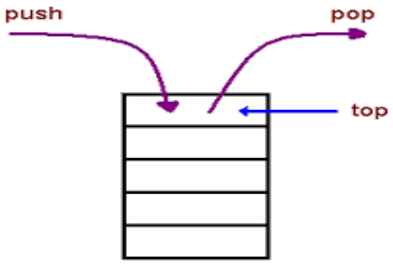
2. Linked list Implementation:

- $p1(x) = 23x^9 + 18x^7 + 41x^6 + 163x^4 + 3$
- $p2(x) = 4x^6 + 10x^4 + 12x + 8$



- Advantages of using a Linked list:
  - save space (don't have to worry about sparse polynomials) and easy to maintain
  - don't need to allocate list size and can declare nodes (terms) only as needed
- Disadvantages of using a Linked list :
  - can't go backwards through the list
  - Can't jump to the beginning of the list from the end.

Stack



- Linear data structure
- Follows LIFO concept
- Addition of new items and the removal of existing items always takes place at the same end. This end is commonly referred to as the "top."

Basically three operations that can be performed on stacks They are

1) inserting an item into a stack **(push)**.
2) deleting an item from the stack **(pop)**.
3) displaying the contents of the stack.

Exceptional Conditions

OverFlow

Attempt to insert an element when the stack is full is said to be overflow.

UnderFlow

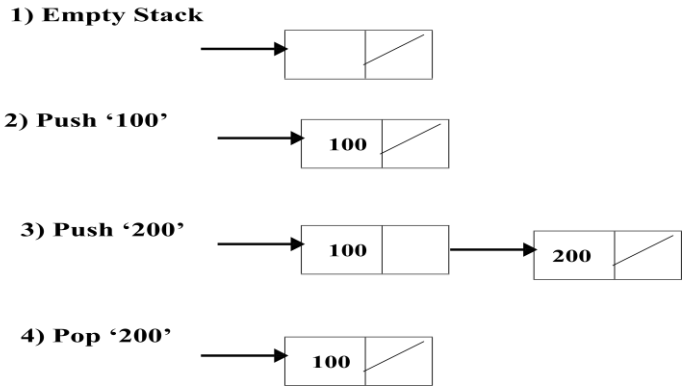
Attempt to delete an element, when the stack is empty is said to be underflow.

Implementation of stack

- Array Implementation of Stack
- Linked List Implementation of Stack
- Array Implementation of Stack
- In this implementation each stack is associated with a pop pointer, which is -1 for an empty stack.
- To push an element X onto the stack, Top Pointer is incremented and then set Stack [Top]
- = X.
- To pop an element, the stack [Top] value is returned and the top pointer is decremented.

Linked List Implementation of Stack

- Push operation is performed by inserting an element at the front of the list.
- Pop operation is performed by deleting at the front of the list.
- Top operation returns the element at the front of the list.



Push

```
void push(int element)
{
if(count == maxnum)
 cout<<"stack is full";
else    {
node *newTop = new node;
if(top == NULL)
 {
 newTop->data = element;
newTop->next = NULL;
top = newTop;
count++;
}
else
{
newTop->data = element;
newTop->next = top;
top = newTop;
 count++;
 }
}
}
```

Pop

```
void pop()
 {
if(top == NULL)
 cout<< "nothing to pop";
else
{
node * old = top;
 top = top->next;
count--; delete(old);
}
}
```

Applications Of Stack

- Some of the applications of stack are :

- Evaluating arithmetic expression
- Balancing the symbols
- Towers of Hannoi
- Function Calls.
- 8 Queen Problem

Queue

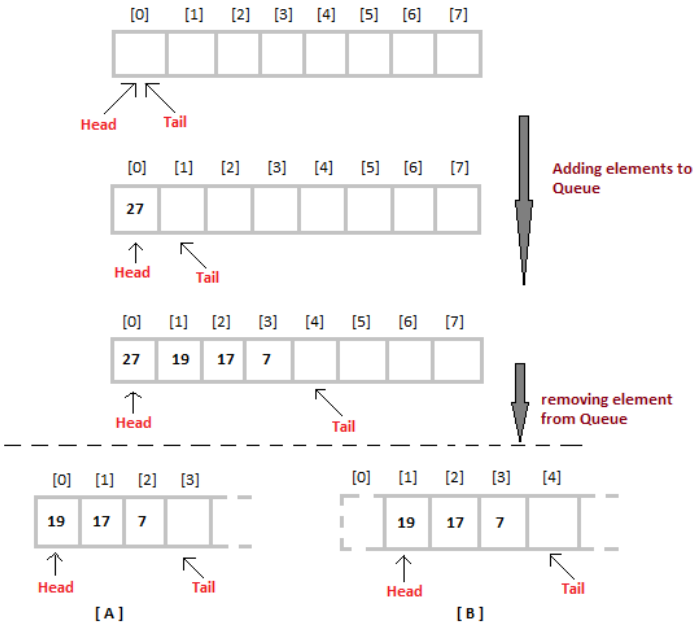
Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called **REAR**(also called tail), and the deletion of exisiting element takes place from the other end called as **FRONT**(also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.

Implementation of queue

1. Array Implementation

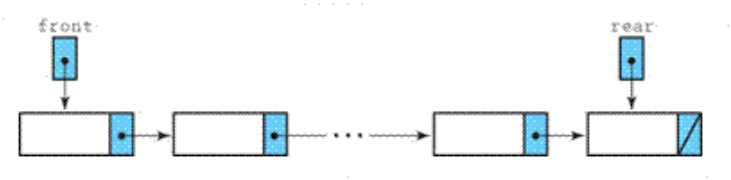
Like a stack, an array is simple to use but has the limitation that the array may overflow. Solution to reuse the array is to wrap around the end of the array.



2. Implementing a queue as a linked list

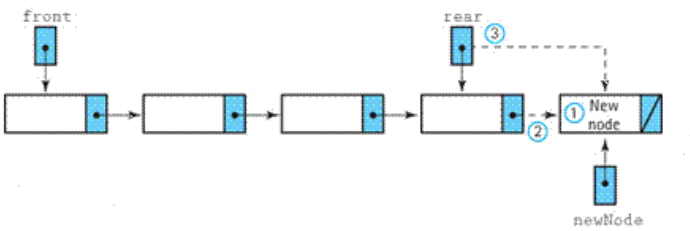
The limitations of using an array for a stack are the same as using an array for queue.For nodes we use the same LLObjectNode class we used for the linked implementation of stacks.

Track front and rear:



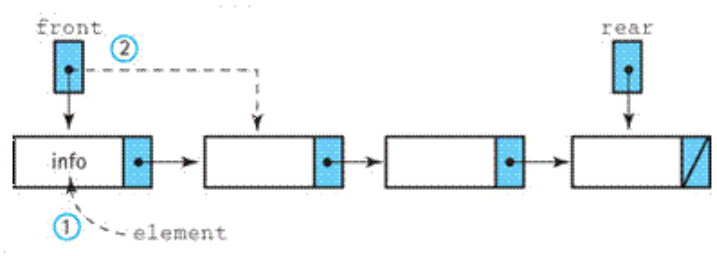
Enqueue at rear so that dequeue is "pop" from front

1. Create a node for the new element
2. Insert the new node at the rear of the queue
3. Update the reference to the rear of the queue
   if front==null then front = rear



Dequeue operation

1. Set element to the information in the front node
2. Remove the front node from the queue
   if the queue is empty
   Set the rear to null return element



Evaluating Arithmetic Expression

Evaluating Arithmetic Expression:

To evaluate an arithmetic expressions, first convert the given infix expression to postfix expression and then evaluate the postfix expression using stack.

1. Infix to Postfix Conversion

Read the infix expression one character at a time until it encounters the delimiter. "#"

Step 1 : If the character is an operand, place it on to the output.

Step 2 : If the character is an operator, push it onto the stack. If the stack operator has a higher or equal priority than input operator then pop that operator from the stack and place it onto the output.

Step 3 : If the character is a left parenthesis, push it onto the stack.

Step 4 : If the character is a right parenthesis, pop all the operators from the stack till it encounters left parenthesis, discard both the parenthesis in the output.

Example: a+b*c+(d*e+f)*g

1) Read ‘a’. place it to output.

a

2) Read ‘+’, push into stack.

a

3) Read ‘b’, place into an output.

ab

4) Read ‘*’, here ‘+’ has low priority than ‘*’, so ‘*’ is push into a stack.

ab

5) Read ‘c’, place into an output.

abc

6) Read ‘+’, here ‘+’ has low priority than ‘*’, so pop ‘*’ & placed into output

abc*

7) In stack ‘+’ has equal priority, so pop ‘+’ and placed in to output.

abc*+

8) Read ‘(‘, place in to stack.

abc*+

9) Read ‘d’, place into an output.

abc*+d

10) Read ‘*’, place into a stack.

abc*+d

11) Read‘e’, place into an output.

abc*+de

12) Read ‘+’, here ‘+’ has low priority than ‘*’, so pop ‘*’ & placed into output

abc*+de*

13) Read‘f’, place into an output.

abc*+de*f

14) Read ‘)’, pop all operators from ‘(‘ to ‘)’ and the popped operators are placed into an output.

abc*+de*f+

15) Read ’*’, here ‘+’ has low priority than ‘*’, so ‘*’ is push into a stack.

abc*+de*f+

16) Read ‘g’ and place it to output.

abc*+de*f+g

17) Now the input string is end, so we pop all the operators from stack and place into output.

abc*+de*f+g*+

2) Evaluating Postfix Expression

Read the postfix expression one character at a time until it encounters the delimiter `#'.

Step 1 : - If the character is an operand, push its associated value onto the stack.

Step 2 : - If the character is an operator, POP two values from the stack, apply the operator to them and push the result onto the stack.

Example: 6523+8*+3+*

	3
TOP→	2

1) First four symbols are pushed on to stack

	5
	6

2) Next '+' is read, so 3 & 2 are popped & their sum 5 is pushed.

	5
TOP→	5
	6

3) Next 8 is pushed to stack.

	8
TOP→	5
	5
	6

4) Next '*' is read, so 8 & 5 are popped and 5*8=40 is pushed to stack

	40
TOP→	5
	6

5) Next '+' is read, so 40 & 5 are popped and 40+5=45 is pushed to stack

	45
TOP→	6

6) Next 3 is pushed to stack.

	3
TOP→	45
	6

7) Next '+' is read, so pop 3 &45

	48
TOP→	6

and their sum 48 is pushed to stack.

8) Next '*' is read. So 6&48 are popped and their product 288 is pushed to stack

TOP→	288