BITS Pilani
Hyderabad Campus

S3 Elementary Data Structures

Dr. Rajib Ranjan Maiti
CSIS Dept, Hyderabad Campus

innovate achieve lead

BITS Pilani
Hyderabad Campus

Data Structures and Algorithms Design (Merged-SEZG519/SSZG519)

S3 Elementary Data Structures

Content of S3

1. Stacks
   1. Stack ADT and Implementation
   2. Applications
2. Queues
   1. Queue ADT and Implementation
   2. Applications
3. Linked List
   1. Notion of positions in lists
   2. List ADT and Implementation

Stacks

Stacks

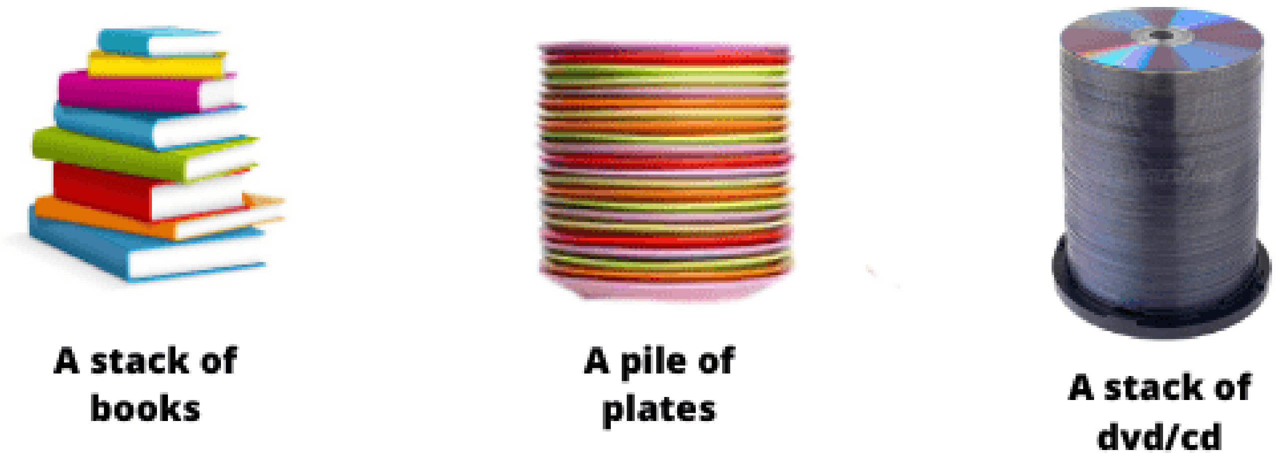


Fig: Realtime examples of Stack

Stack

- Stack
  - A container, where objects are inserted and deleted according to LIFO

- List of operations on stack:
  - push(): Inserts an object into the top of the stack
  - pop(): Removes the object from the top of the stack; an error occurs if the stack is empty
  - size(): Returns the number of objects in the stack
  - isEmpty(): Return True if the stack is empty
  - top(): Return the object at the top of the stack, but does not remove it; error, if the stack is empty

Stack (push and pop)

Algorithm push(obj):

if size( ) = N **then**

error: stack is full

t ← t+1

S[t] ← obj

Algorithm pop( ):

if isEmpty( ) **then**

error: stack is empty

obj ← S[t]

S[t] ← **NULL**

t ← t-1

return obj

Other operations on stack: size(), isEmpty(), and top()

```
Algorithm size( ):
        return t+1
```

```
Algorithm isEmpty( ):
        if t = 0 then
                return TRUE
        else
                return FALSE
```

```
Algorithm top( ):
        if t = 0 then
                return NULL
        else
                return S[t]
```

Applications of Stack
(Expression Evaluation: Infix-Prefix-Postfix)

Arithmetic expression: consists of operands and operators

Notations for arithmetic expression:

- **Infix**: <operand> <operator> <operand>
- **Prefix**: <operator> <operand> <operand>
- **Postfix**: <operand> <operand> <operator>

- **Why so many representations?**
- Infix notation can be converted to equivalent Prefix, Postfix notations
- Prefix and Postfix notations are parenthesis free and faster than infix.

Applications of Stack
(Expression Evaluation: Infix-Prefix-Postfix)

Infix:

((A+((B^C)-D))*(E-(A/C)))

Prefix equivalent to Infix:

* + A - ^ BCD – E/AC

Postfix equivalent to Infix:

ABC ^ D - + EAC / - *

Applications of Stack
(Expression Evaluation: Infix-Prefix-Postfix)

```
Algorithm infixtopostfix(input, len):
        input[len + 1] ← ')'               // insert ')' as the last element in the array
        Push ('(')                         // push '(' into the stack
        while isEmpty() = FALSE do         // stack is not empty
                if input[i] is operand then
                        print input[i]
                else if input[i] = ')'
                        while x← POP( ) ≠ '(' do
                                print x
                else if Priority(input[i]) <= Priority(TOP(s)) //stack contains only operators
                        while Priority(input[i]) <= Priority(TOP(s)) do
                                x ← POP( )
                                print x
                         Push(input[i])            //input[i] is an operator
                else
                        Push(input[i])
                i←i+1
```

Applications of Stack
(Expression Evaluation: Infix-to-Postfix)

Input	Stack	Output
(A+B)^C-(D*E)/F		
(A+B)^C-(D*E)/F**)**	(	
(A+B)^C-(D*E)/F)	((	
(**A**+B)^C-(D*E)/F)	((	A
(A**+**B)^C-(D*E)/F)	((+	A
(A+**B**)^C-(D*E)/F)	((+	AB
(A+B**)**^C-(D*E)/F)	(	AB+
(A+B)**^**C-(D*E)/F)	(^	AB+
(A+B)^**C**-(D*E)/F)	(^	AB+C
(A+B)^C**-**(D*E)/F)	(-	AB+C^
(A+B)^C-**(**D*E)/F)	(- (	AB+C^
(A+B)^C-(**D***E)/F)	(- (	AB+C^D

Applications of Stack
(Expression Evaluation: Infix-to-Postfix)

Input	**Stack**	**Output**
(A+B)^C-(D*****E)/F)	(- (*	AB+C^D
(A+B)^C-(D***E**)/F)	(- (*	AB+C^DE
(A+B)^C-(D*E**)**/F)	(-	AB+C^DE*
(A+B)^C-(D*E)**/**F)	(- /	AB+C^DE*
(A+B)^C-(D*E)/**F**)	(- /	AB+C^DE*F
(A+B)^C-(D*E)/F)	EMPTY	AB+C^DE*F/-

Applications of Stack

(Expression Evaluation: Postfix Evaluation Algorithm)

```
Algorithm postfixeval(input, len):
        input[len + 1] ← '#'
        i←1
        while (item←input[i]) ≠ '#' do
                if item is operand then
                        push(item)
                else
                        op←item
                        y←pop( )
                        x←pop( )
                        z←x op y
                        Push(z)
                i←i+1
        answer←pop( )
        return answer
```

Applications of Stack
(Expression Evaluation: Postfix Evaluation Example)

Input	Stack
AB+C^DE*F/-	
23+5^67*3/-	
23+5^67*3/-	2
2**3**+5^67*3/-	23
23**+**5^67*3/-	5
23+**5**^67*3/-	5 5
23+5**^**67*3/-	0
23+5^**6**7*3/-	0 6
23+5^6**7***3/-	0 6 7
23+5^67*****3/-	0 42
23+5^67***3**/-	0 42 3
23+5^67*3**/**-	0 14
23+5^67*3/**-**	-14

Infix: (A+B)^C-(D*E)/F

(A+B)^C-(D*E)/F

= (2+3)^5-(6*7)/3

= 5^5- (6*7)/3

= 5^5- 42/3 = 0 – 42/3

=0-14 = -14

Postfix: AB+C^DE*F/-

Applications of Stack
(Expression Evaluation: Infix-to-Prefix Algorithm)

```
Algorithm infixtoprefix(reverse(input), len):
        input[len + 1] ← '('
        Push(')')
        while isEmpty( ) = FALSE do
                if input[i] is operand then
                        output[j++] ← input[i]
                else if input[i] = '('
                        while x← POP( ) ≠ ')' do
                                output[j++] ← x
                else if Priority (input[i]) <= Priority (Top(s))
                         while Priority (input[i]) <= Priority (Top(s)) do
                                x ← Pop( )
                                output[j++] ← x
                        Push (input[i])
                else
                        Push(input[i])
                i←i+1
        print(reverse(output))
```

Applications of Stack
(Expression Evaluation: Infix-to-Prefix Exact)

innovate achieve lead

Input	Stack	Output
(A+B)^C-(D*E)/F		
F/)E*D(-C^)B+A((	)	
F/)E*D(-C^)B+A((	)	F
F**/**)E*D(-C^)B+A((	)/	F
F/**)**E*D(-C^)B+A((	)/)	F
F/)**E***D(-C^)B+A((	)/)	FE
F/)E*****D(-C^)B+A((	)/)*	FE
F/)E***D**(-C^)B+A((	)/)*	FED
F/)E*D**(**-C^)B+A((	)/	FED*
F/)E*D(**-**C^)B+A((	)-	FED*/
F/)E*D(-**C**^)B+A((	)-	FED*/C
F/)E*D(-C**^**)B+A((	)-^	FED*/C

Applications of Stack
(Expression Evaluation: Infix-to-Prefix Example)

Input	Stack	Output
F/)E*D(-C**^**)B+A((	)-^	FED*/C
F/)E*D(-C^**)**B+A((	)-^)	FED*/C
F/)E*D(-C^)**B**+A((	)-^)	FED*/CB
F/)E*D(-C^)B**+**A((	)-^)+	FED*/CB
F/)E*D(-C^)B+**A**((	)-^)+	FED*/CBA
F/)E*D(-C^)B+A**(**(	)-^	FED*/CBA+
F/)E*D(-C^)B+A(**(**		FED*/CBA+^-

Reverse(FED*/CBA+^-) = -^+ABC/*DEF

Applications of Stack
(Expression Evaluation: Prefix Evaluation Algorithm)

```
Algorithm prefixeval(reverse(input), len):
        input[len + 1] ← '#'
        i←1
        while (item←input[1]) ≠ '#' do
                if item is operand then
                        push(item)
                else
                        op←item
                        x←pop( )
                        y←pop( )
                        z←x op y
                        push(z)
                i←i+1
        answer←pop( )
        return answer
```

Applications of Stack
(Expression Evaluation: Prefix Evaluation Example)

Input	Stack
FED*/CBA+^-	
376*/532+^-	
376*/532+^-	3
3**7**6*/532+^-	3 7
37**6***/532+^-	3 7 6
376*****/532+^-	3 42
376*/532+^-	14
376*/**5**32+^-	14 5
376*/5**3**2+^-	14 5 3
376*/53**2**+^-	14 5 3 2
376*/532**+**^-	14 5 5
376*/532+**^**-	14 0
376*/532+^**-**	-14

Infix: (A+B)^C-(D*E)/F

(A+B)^C-(D*E)/F

= (2+3)^5-(6*7)/3

= 5^5- (6*7)/3

= 5^5- 42/3 = 0 – 42/3

=0-14 = -14

Prefix: -^+ABC/*DEF

Reverse(Prefix):

FED*/CBA+^-

Applications of Stack
(Expression Evaluation: Postfix-to-Infix Algorithm)

```
Algorithm postfixtoinfix(input, len):
        input[len + 1] ← '#'
        i←1
        while (item←input[1]) ≠ '#' do
                if item is operand then
                        push(item)
                else
                        op←item
                        y←pop( )
                        x←pop( )
                        z←(x op y)
                        push(z)
                i←i+1
        answer←pop( )
        return answer
```

Applications of Stack
(Expression Evaluation: Postfix-to-Infix Example)

Input	Stack
AB+C^DE*F/-	A
AB+C^DE*F/-	AB
AB+C^DE*F/-	(A+B)
AB+C^DE*F/-	(A+B)C
AB+C^DE*F/-	((A+B)^C)
AB+C^DE*F/-	((A+B)^C)D
AB+C^DE*F/-	((A+B)^C)DE
AB+C^DE*F/-	((A+B)^C)(D*E)
AB+C^DE*F/-	((A+B)^C)(D*E)F
AB+C^DE*F/-	((A+B)^C)((D*E)/F)
AB+C^DE*F/-	(((A+B)^C)-((D*E)/F))

Postfix: AB+C^DE*F/-

Infix: ((A+B)^C)-((D*E)/F)

Applications of Stack
(Expression Evaluation: Prefix-Infix Algorithm)


```
Algorithm prefixtoinfix(reverse(input), len):
        input[len + 1] ← '#'
        i←1
        while (item←input[i]) ≠ '#' do
                if item is operand then
                        push(item)
                else
                        op←item
                        x←pop( )
                        y←pop( )
                        z←(x op y)
                        push(z)
                i←i+1
        answer←pop( )
        return answer
```

Applications of Stack
(Expression Evaluation: Prefix-Infix Example)

Input	Stack
FED*/CBA+^-	F
F**E**D*/CBA+^-	F E
FE**D***/CBA+^-	F E D
FED*****/CBA+^-	F (D*E)
FED***/**CBA+^-	((D*E)/F)
FED*/**C**BA+^-	((D*E)/F) C
FED*/C**B**A+^-	((D*E)/F) C B
FED*/CB**A**+^-	((D*E)/F) C B A
FED*/CBA**+**^-	((D*E)/F) C (A+B)
FED*/CBA+**^**-	((D*E)/F) ((A+B)^C)
FED*/CBA+^**-**	(((A+B)^C)-((D*E)/F))

Prefix: -^+ABC/*DEF

Reverse(Prefix):

FED*/CBA+^-

Infix: ((A+B)^C)-((D*E)/F)

Applications of Stack
(Stack machines)

- Stack-based machines and Register-based machines
- Stack-based machines:

A = B * C – A

Postfix: A B C * A – =

			C		A		
		B	B	B*C	D	D-A	
	A	A	A	A	A	A	
Empty	Push A	Push B	Push C	MUL	Push A	SUB	POP
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)

Applications of Stack
(Stack machines)

- Stack-based machines and Register-based machines
- Stack-based machines:

Advantages:

- Higher code density.

- Fewer registers required

Disadvantages:

- Slower access to variables

Applications of Stack
(Stack machines)

- Stack-based machines and Register-based machines
- Register-based machines:

A = B * C – A

```
mov r1, B
mul r1, C
sub r1, A
```

Advantages:

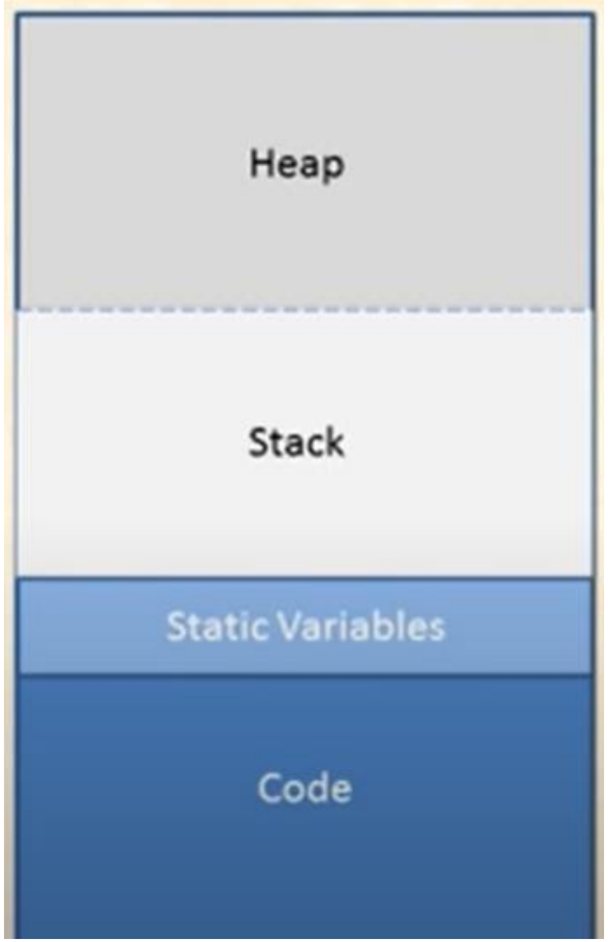
- Faster access to variables

- Fewer memory operations

Disadvantages:

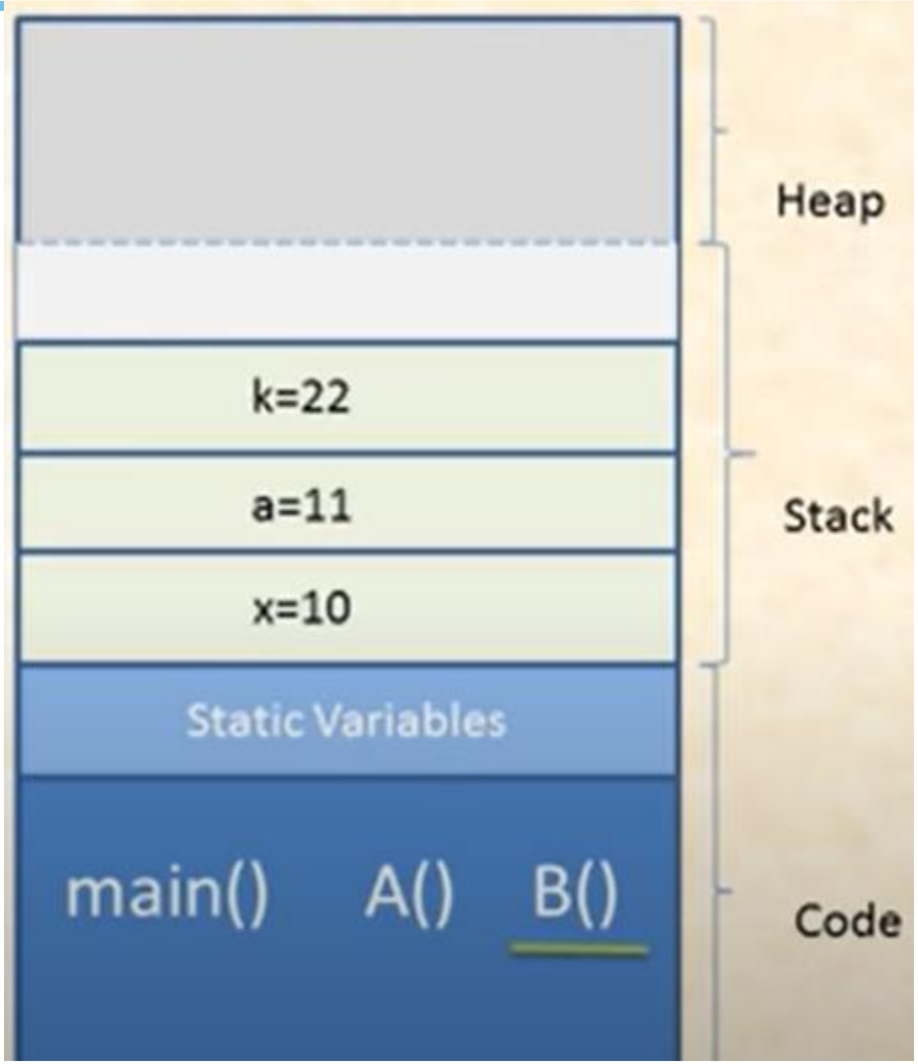
- More registers for dense code

Applications of Stack (Recursion)



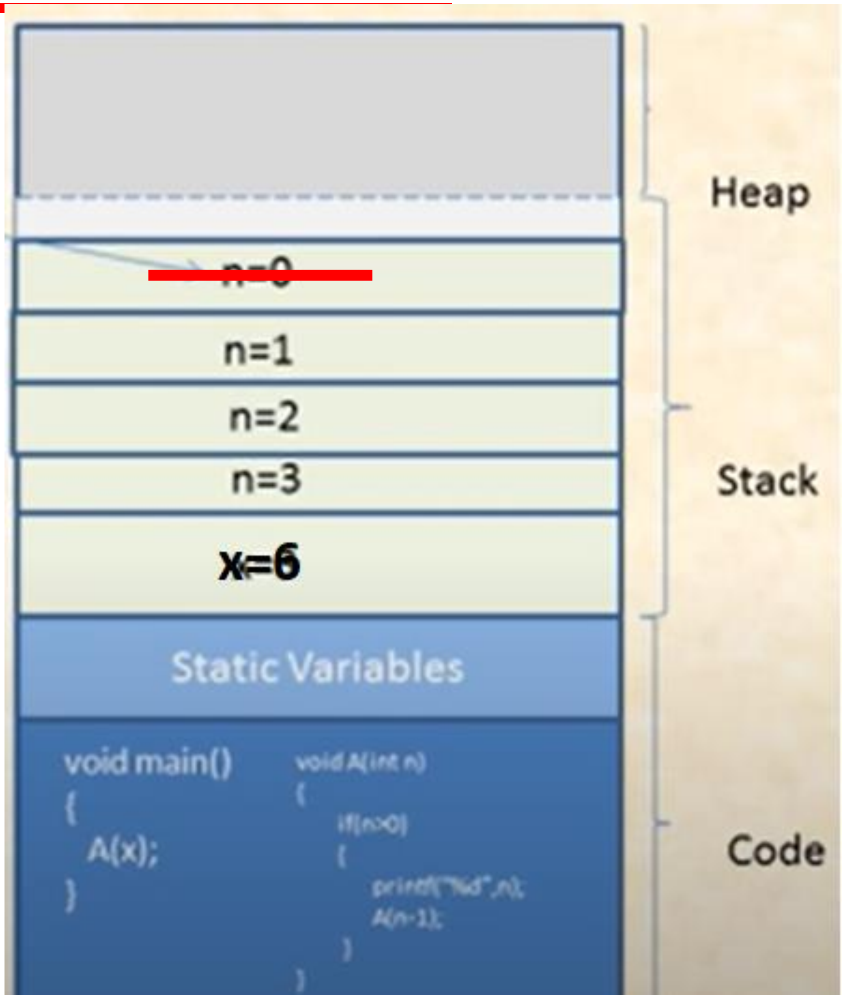
Applications of Stack (Recursion)

```
void B(int k){
      printf("%d",k);
}
void A(int a){
      B(a*2);
      printf("%d",a);
}
void main(){
      int x=10;
      A(x+1);
}
```

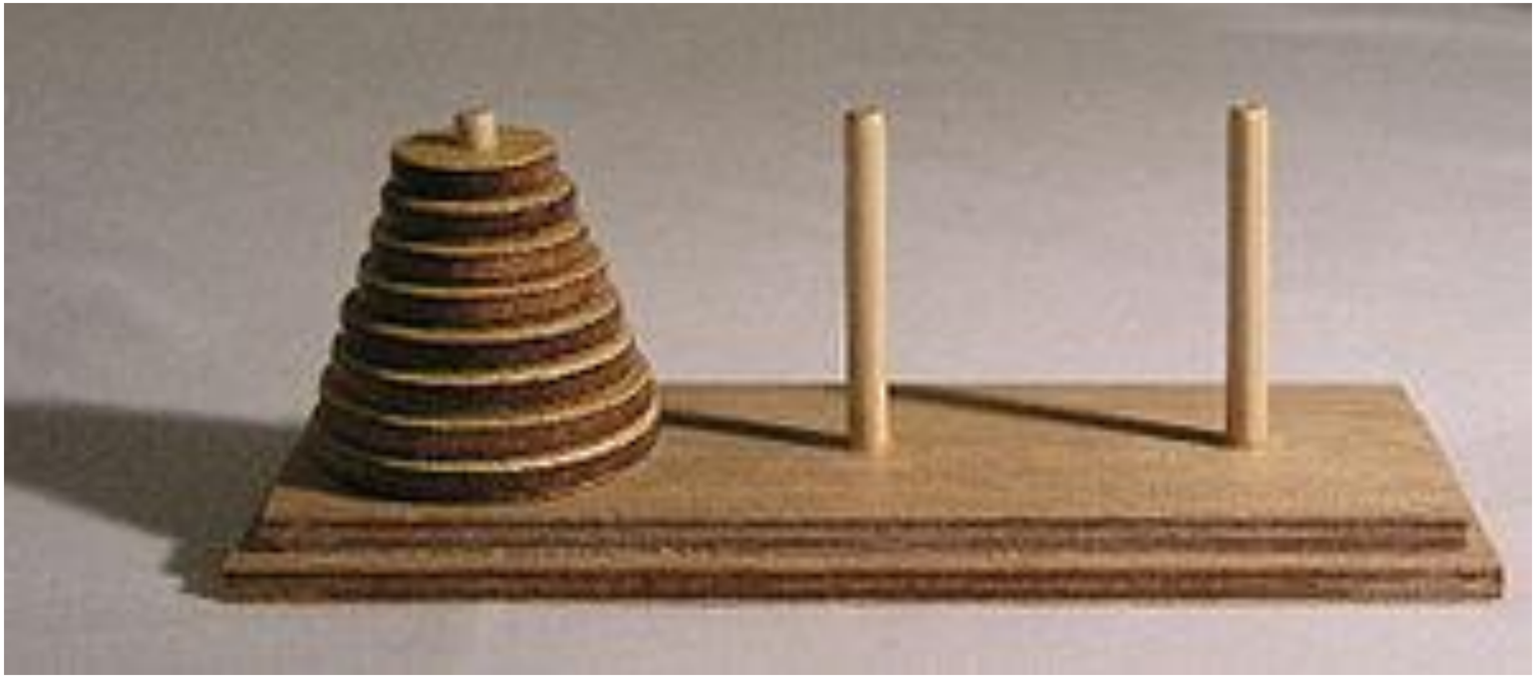


Applications of Stack (Recursion)

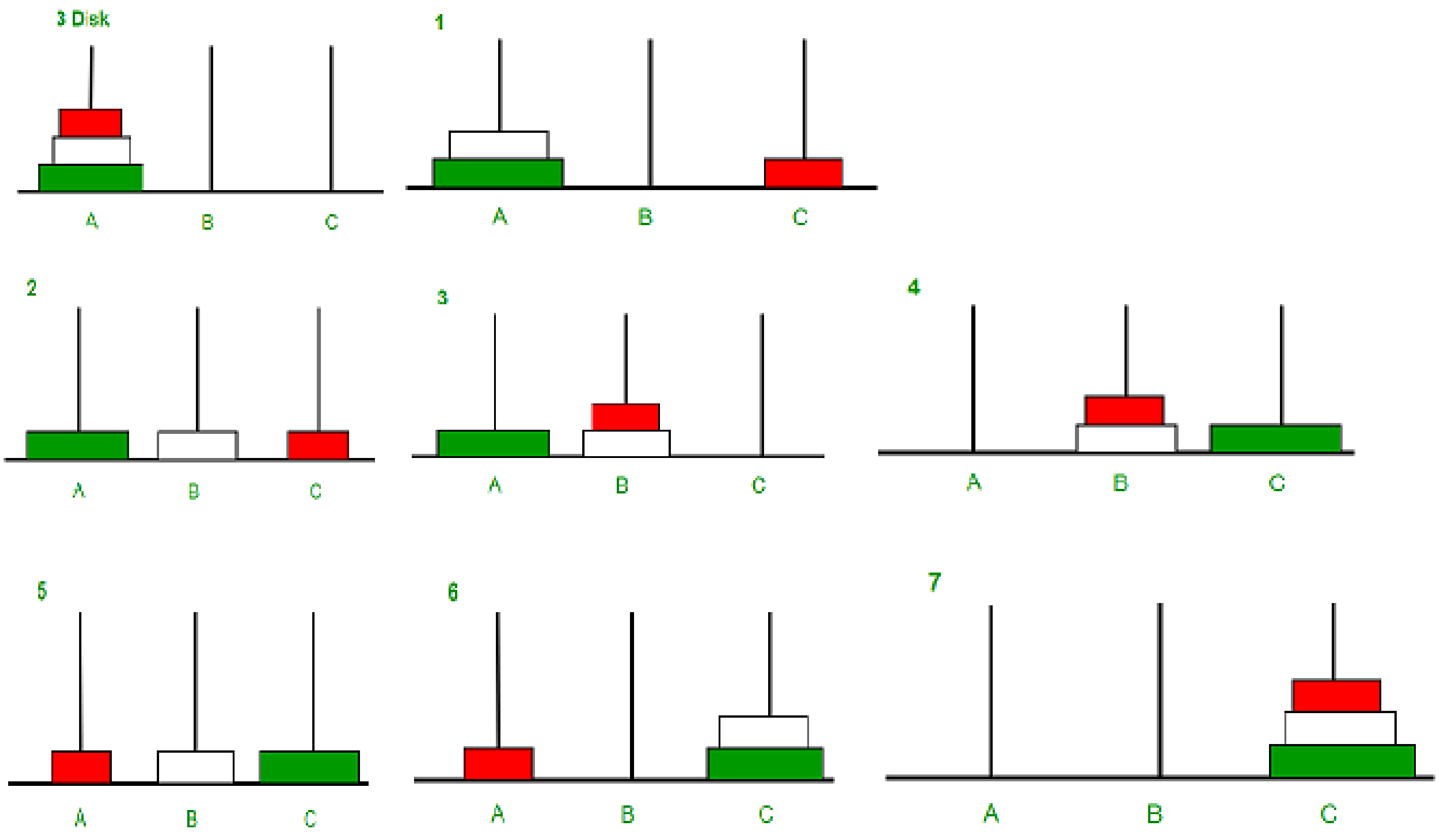
```c
void A(int n){
    if(n>0){
            printf("%d",n);
            A(n-1);
    }
}
void main(){
    int x=6;
    A(x/2);
}
```



Applications of Stack
(Tower of Hanoi)



Applications of Stack
(Tower of Hanoi: The Problem)



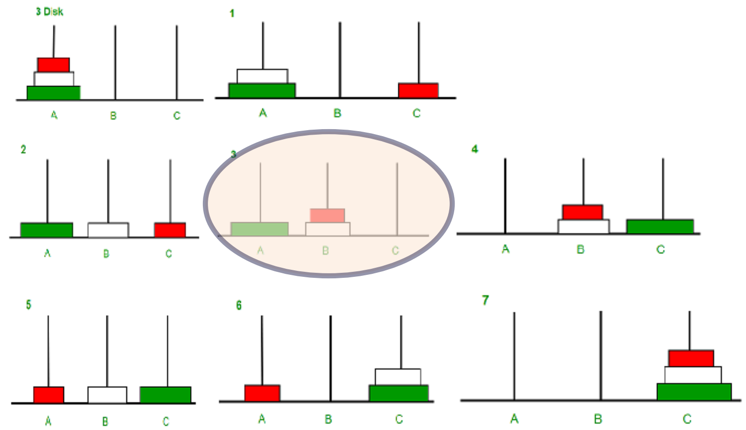
Applications of Stack
(Tower of Hanoi: Solution Approach)

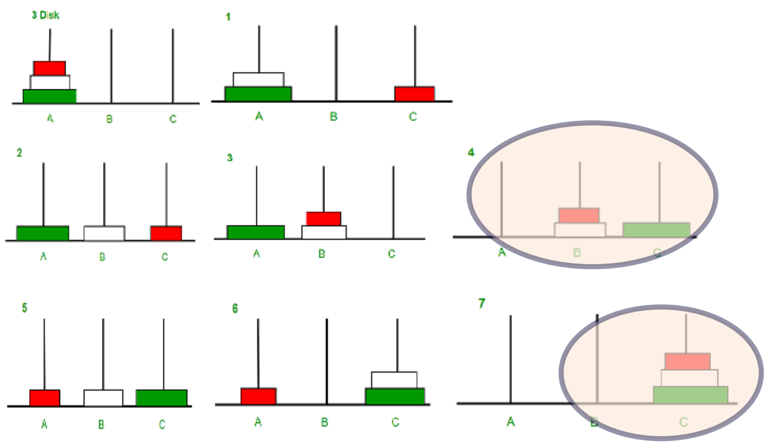
Move n-1 discs from A to B using C

Move a disc from A to C

Move n-1 discs from B to C using A

Total Moves = $(2^n) - 1$

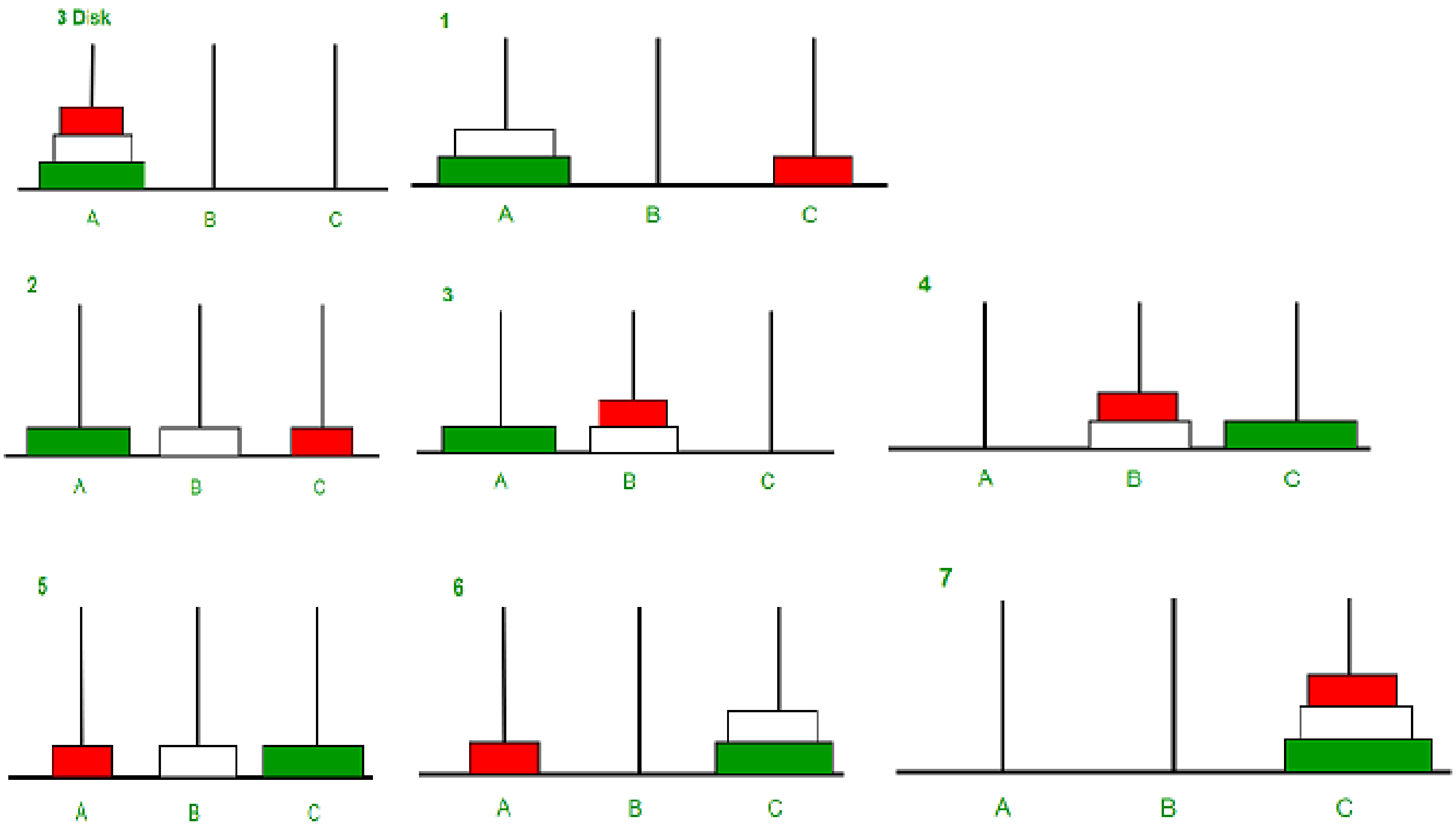




Applications of Stack
(Tower of Hanoi: Algorithm)

```
#include<stdio.h>
void TOH(int n,int A,int B,int C){//move n from A to C using B
    if(n>0){
        TOH(n-1,A,C,B);          //move n-1 from A to B using C
        printf("Move %dth disc from %d to %d\n",n,A,C); //move 1 from A C
        TOH(n-1,B,A,C);          //move n-1 from B to C using A
    }
}
void main(){
    int n=3;
    int A=1,B=2,C=3;
    TOH(n,A,B,C);
}
```

Applications of Stack
(Tower of Hanoi)

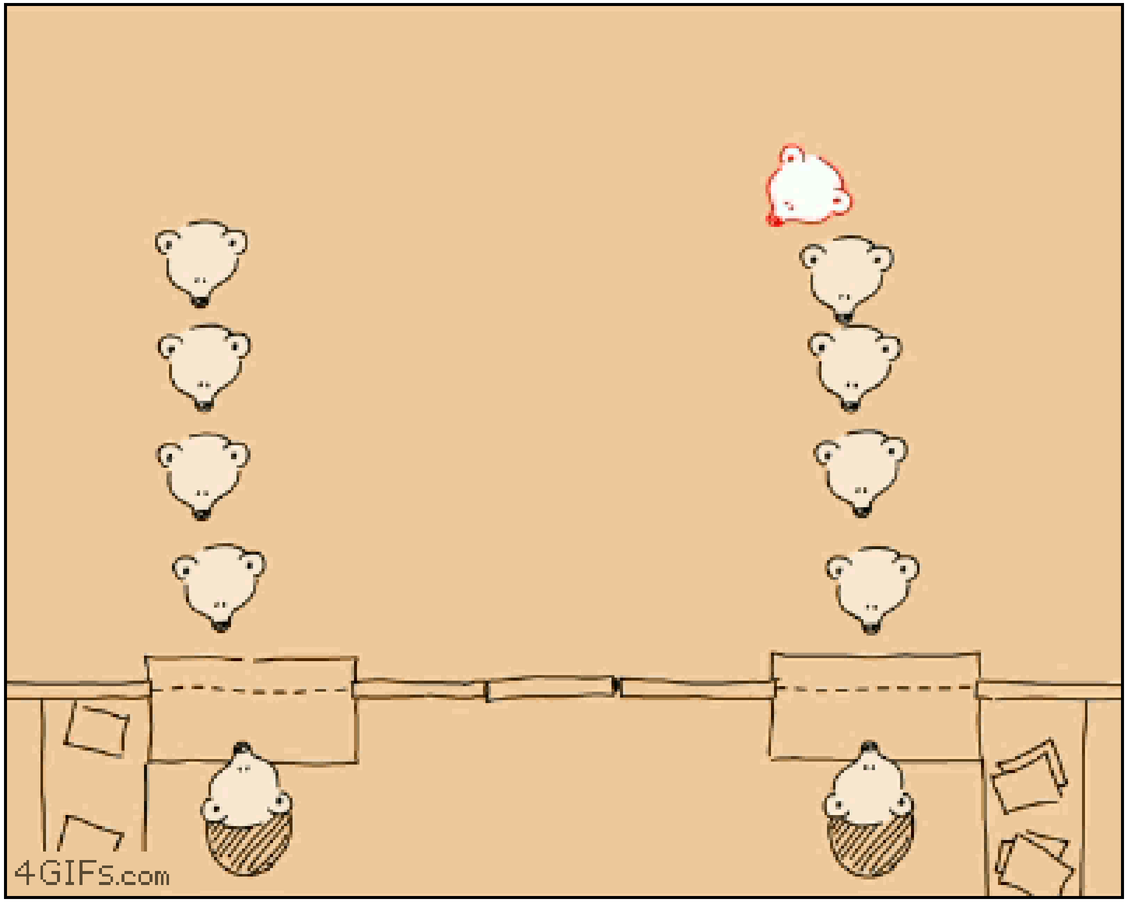


Applications of Stack
(Tower of Hanoi)

```
Iterative:

for i = 1 to (2^n)-1
        if i%3 = 1 then
                Move top disk from A to C
        if i%3 = 2 then
                Move top disk from A to B
        if i%3 = 0 then
                Move top disk from B to C
```

Queue



Queue

- Close "cousin" of the stack
- A queue is a container of objects that are inserted and removed according to the first-in first-out (FIFO) principle.
- We usually say that elements enter the queue at the rear and are removed from the front.
- List of Operations on Queue:
  - enqueue(o): Insert object o at the rear of the queue.
  - dequeue( ): Remove and return from the queue the object at the front; an error occurs if the queue is empty.
  - size( ): Return the number of objects in the queue.
  - isEmpty( ): Return a Boolean value indicating whether queue is empty.
  - front( ): Return, but do not remove, the front object in the queue, an error occurs if the queue is empty.

Queue (enqueue and dequeue)

```
Algorithm enqueue(o):
        if size( ) = N then
                indicate that a queue-full error has occurred
                return
        r←r+1
        Q[r]←o
        if f = 0 then
                f = 1
Algorithm dequeue( ):
        if isEmpty( ) then
                indicate that a queue-empty error has occurred
                return NULL
        e←Q[f]
        Q[f]← NULL
        if f = r then
                f ← 0
                r ← 0
        else
                f←f+1
        return e
```

Queue (size, isempty, front)

Algorithm size( ):

    return r – f +1

Algorithm isempty( ):

    if f = 0 **then**

        return TRUE

    else

        return FALSE

Algorithm front( ):

    if f = 0 **then**

        return NULL

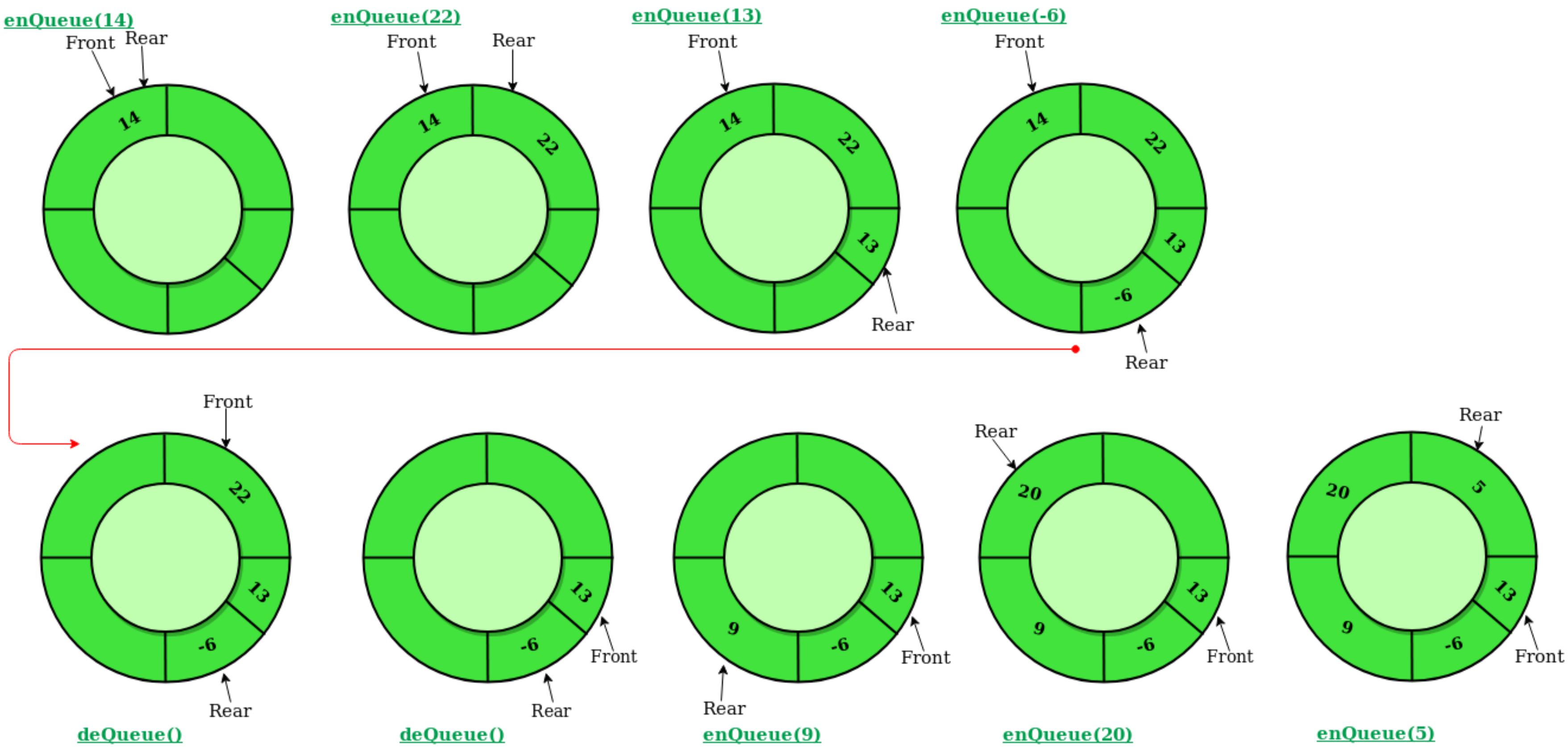
    else

        return Q[f]

Queue (Applications)

1. Task scheduling
2. Print spooling
3. Breadth-First Search (BFS)

Circular Queue



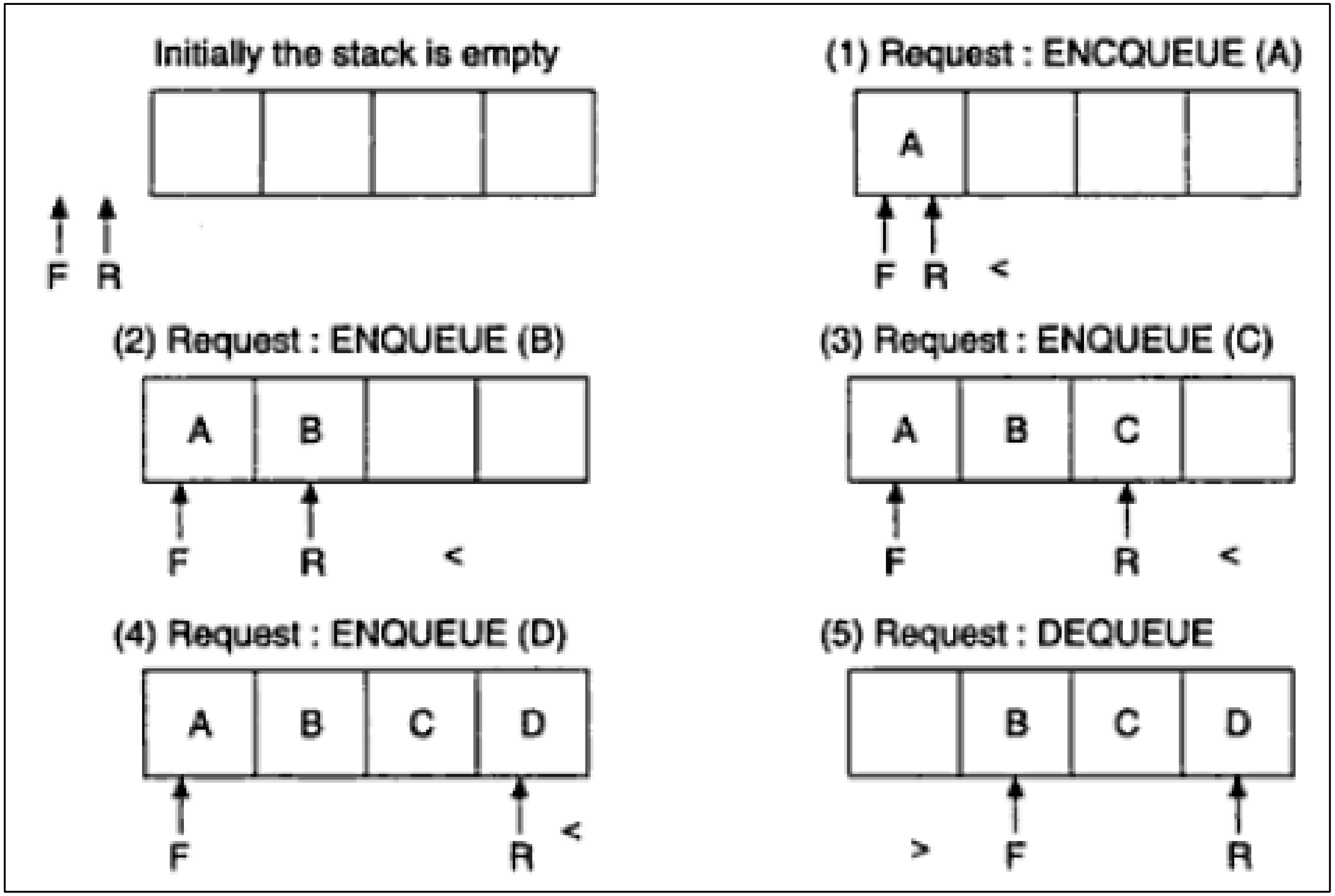
Circular Queue (enqueue)

```
Algorithm enqueue(o):
        if f = 0 then
                f ← 1
                r ← 1
        else
                if r mod n + 1 =  f then
                        indicate that a queue-full error has occurred
                        return
                else
                        r ← r mod n + 1
        Q[r] ← o
```

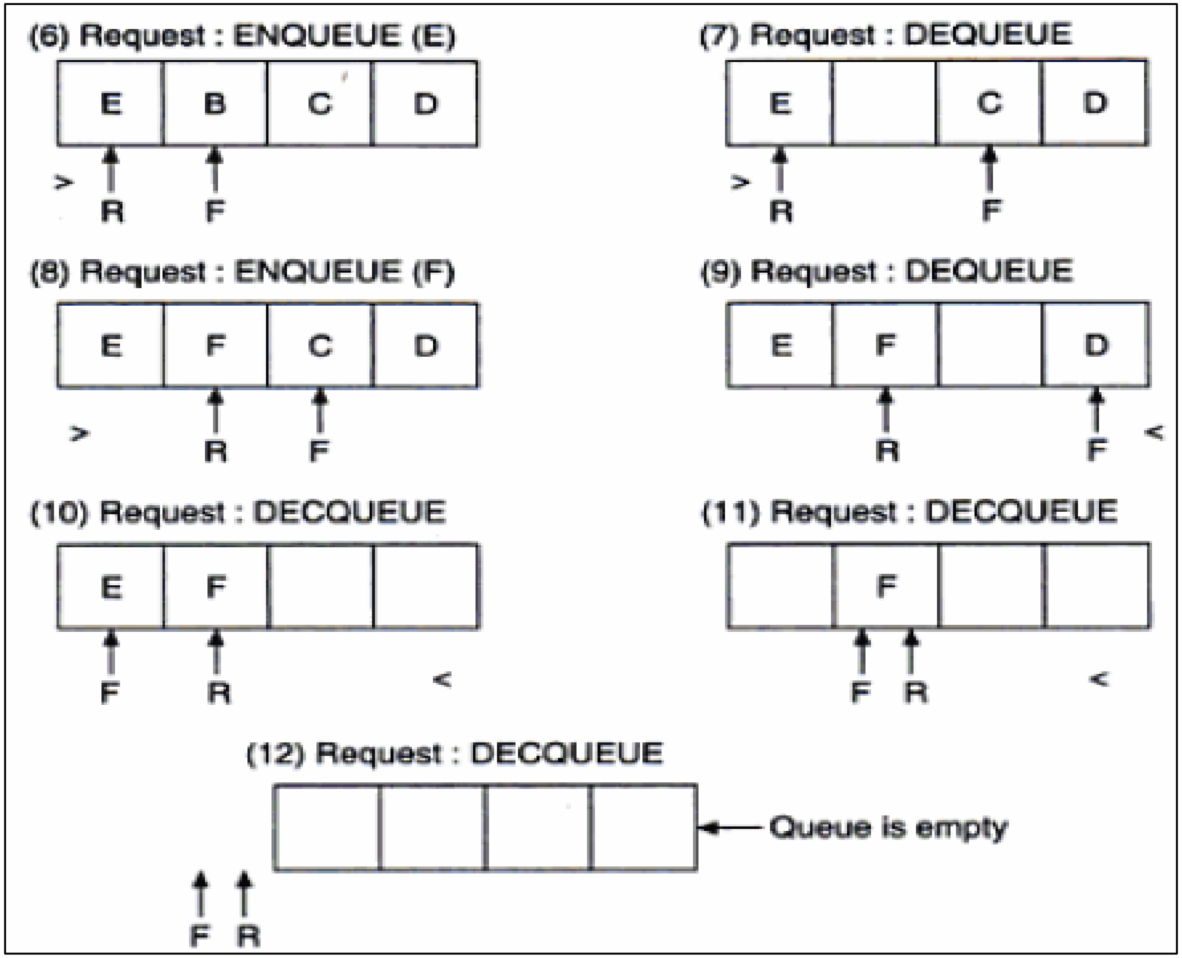
Circular Queue (dequeue)

```
Algorithm dequeue(o):
        if f = 0 then
                indicate that a queue-empty error has occurred
                return NULL
        e ← Q[f]
        Q[f] ← NULL
        if f = r then
                f ← 0
                r ← 0
        else
                f ← f mod n +1
        return e
```

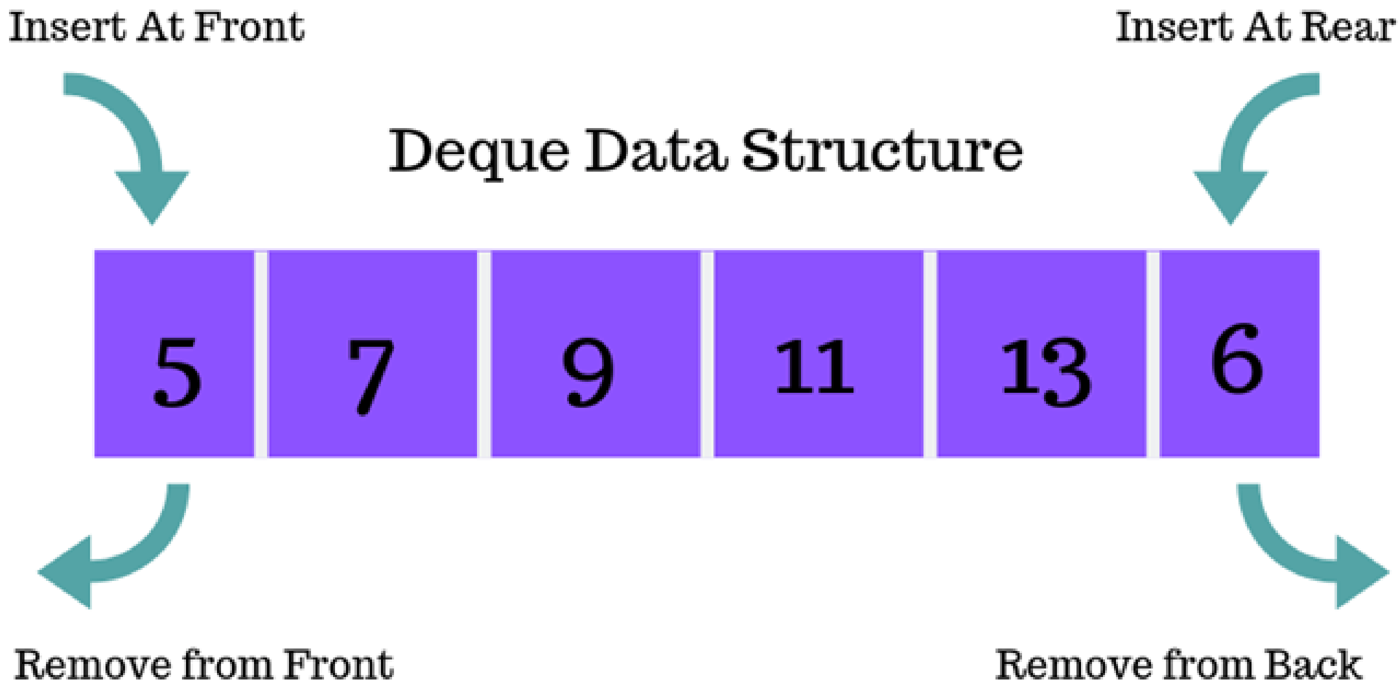
Circular Queue (example)



Circular Queue (example)



DQueue/Double Ended Queue



Dqueue (enqueue)

```
Algorithm enqueuerear(o):
        if r mod n + 1 =  f then
                indicate that a queue-full error has occurred
                return
        else
                if f = 0 and r = 0 then
                        f ← 1
                        r ← 1
                else
                        r ← r mod n + 1
        Q[r] ← o
```

Dqueue (enqueue)

```
Algorithm enqueuefront(o):
        if r mod n + 1 =  f then
                    indicate that a queue-full error has occurred
                    return
        else
              if f = 0 and r = 0 then
                    f ← 1
                    r ← 1
              else if f = 1 then
                    f ← n
              else
                    f ← f - 1
        Q[f] ← o
```

Dqueue (deque)

Algorithm dequeuefront(o):

if f = 0 **then**

indicate that a queue-empty error has occurred

return NULL

e $\leftarrow$ Q[f]

Q[f] $\leftarrow$ **NULL**

if f = r **then**

f $\leftarrow$ 0

r $\leftarrow$ 0

else

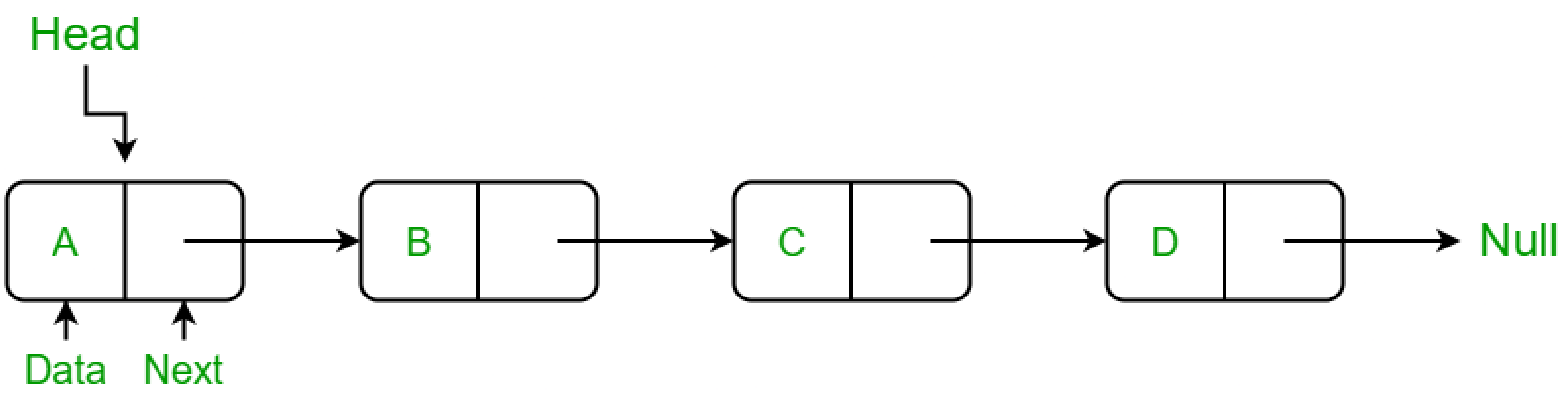
f $\leftarrow$ f mod n +1

return e

Dqueue (deque)

```
Algorithm dequeuerear(o):
        if f = 0 then
                indicate that a queue-empty error has occurred
                return NULL
        e ← Q[r]
        Q[r] ← NULL
        if f = r then
                f ← 0
                r ← 0
                return e
        else
                if r = 1 then
                        r ← n
                else
                        r ← r - 1
        return e
```

Linked List



Linked List

- A collection of finite number of nodes where linear order is maintained by means of links or pointer to other node.
- List of Operations on Linked List:
  - insertatfirst(o): Insert object o at the starting of the linked list.
  - insertatlast(o): Insert object o at the last of the linked list.
  - insertatany(o,key): Insert object o at the key position of the linked list.
  - deleteatfirst(): Delete object at the starting of the linked list.
  - deleteatlast(): Delete object at the last of the linked list.
  - deleteatany(key): Delete object at the key position of the linked list.
  - copy(link1,link2): Copy linked list 1 to linked list 2.
  - merge(link1,link2): Attach linked list2 behind linked list 1
  - search(key): search whether key is in linked list or not

Linked List

insertatfirst(header, Object o)

1. newnode ← Getnode( )
2. newnode.data ← o
3. newnode.next ← header
4. header ← new

Linked List

insertatlast(header, Object o)

1. newnode ← Getnode( )
2. newnode.data ← o
3. ptr ← header
4. **while** ptr.next $\neq$ NULL **do**
    a. ptr ← ptr.next
5. ptr.next ← newnode
6. newnode.next←NULL

Linked List

insertatany(Object o, Object key): Insert object o after the key position of the linked list.

1. ptr ← header
2. **if** ptr = NULL **then**
   a. Print "List Empty"
   b. Exit
3. **while** ptr ≠ NULL **do**
   a. **if** ptr.data = key **then**
      i. newnode ← Getnode( )
      ii. newnode.data ← o
      iii. newnode.next ← ptr.next
      iv. ptr.next←newnode
      v. **Return**
4. Print "Data not found"
5. **Return**

Linked List

deleteatfirst(): Delete object at the starting of the linked list.

1. ptr ← header
2. **if** ptr = NULL **then**
   a. Print “List Empty”
   b. Exit
3. retvalue ← ptr.data
4. header←ptr.next
5. Free(ptr)
6. **return** retval

Linked List

deleteatlast(): Delete object at the last of the linked list.

1. ptr ← header
2. **if** ptr = NULL **then**
   a. Print “List Empty”
   b. Exit
3. **if** ptr.next = NULL **then**
   a. retval←ptr.data
   b. Free(ptr)
   c. Return retval
4. **while** (ptr.next).next ≠ NULL **do**
   a. ptr ← ptr.next
5. ptr1 ← ptr.next
6. retval←ptr1.data
7. Free(ptr1)
8. ptr.next←NULL
9. **Return** retval

Linked List

deleteatany(Object key): Delete object at the key position of the linked list.

1. prev ← header
2. **if** prev = NULL **then**
   a. Print “List Empty”
   b. Exit
3. **if** prev.data = key **then**
   a. retval ← prev.data
   b. header ← prev.next
   c. Free(prev)
   d. **Return** retval
4. curr ← prev.next
5. **while** curr ≠ NULL **do**
   a. **if** curr.data = key then
      i. retval ← curr.data
      ii. prev.next ← curr.next
      iii. Free(curr)
      iv. **Return** retval
   b. prev ← curr
   c. curr ← curr.next
6. Print “Data not found”
7. **Return** NULL

Linked List

merge(header1,header2,header): Attach list2 behind list1

1. ptr $\leftarrow$ header1
2. **while** ptr.next $\neq$ NULL **do**
   a. ptr $\leftarrow$ ptr.next
3. ptr.next $\leftarrow$ header2
4. header $\leftarrow$ header1
5. **return** header

Linked List

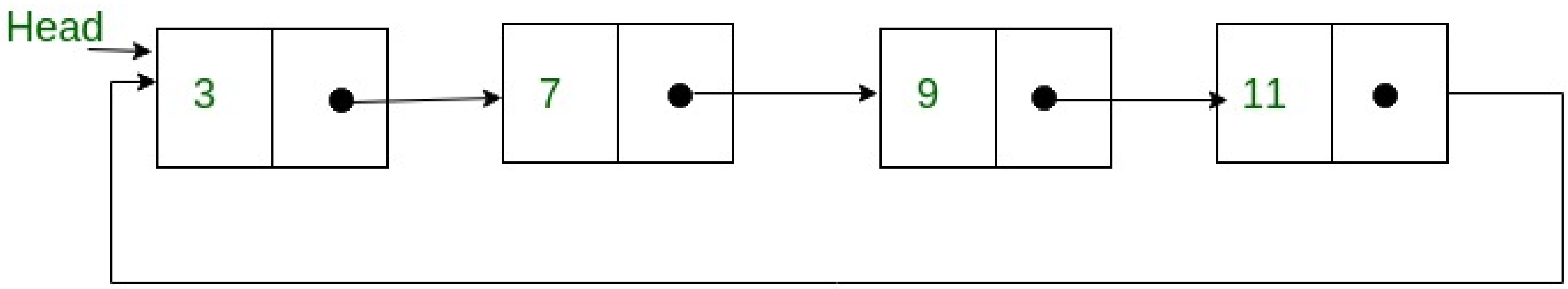
search(Object key): search whether key is in linked list or not

1. ptr ← header
2. **while** ptr.next ≠ NULL **do**
   a. **if** ptr.data = key **then**
      return ptr
3. **return** NULL

Linked List (Applications)

1. Dynamic memory management
2. Polynomial calculations
3. Arithmetic on long numbers
4. Implementing graph, tree, queue, stack
5. Memory representation

Circular Linked List



Circular Linked List

Advantages:

1. Efficient Accessibility

Example: Find the number of elements higher than or equal to each key in the list.

Input: 11 → 4 → 23 → 43 → 5

Output: 2,4,1,0,3

2. Solution to Null link problem

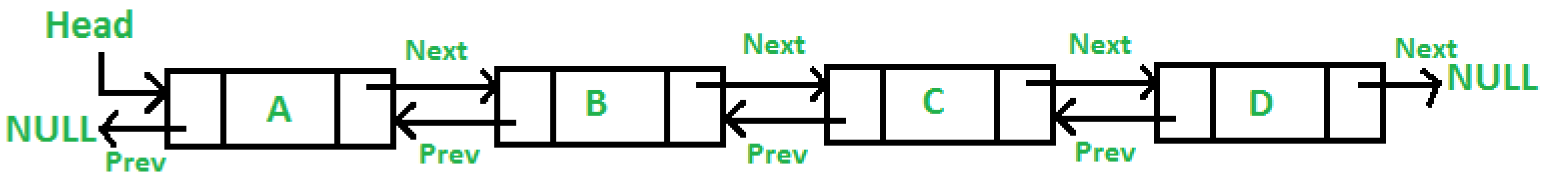
Null value in the link field may create some problem during the execution of the program if a proper care is not taken.

Disadvantages:

1. Infinite loop: Don’t know where to stop!!

Possible solution: Keep special header node, that has no data.

Doubly Linked List



Doubly Linked List

- List of Operations on Linked List:
  - insertatfirst(o): Insert object o at the starting of the linked list.
  - insertatlast(o): Insert object o at the last of the linked list.
  - insertatany(o,key): Insert object o at the key position of the linked list.
  - deleteatfirst(): Delete object at the starting of the linked list.
  - deleteatlast(): Delete object at the last of the linked list.
  - deleteatany(key): Delete object at the key position of the linked list.
  - copy(link1,link2): Copy linked list 1 to linked list 2.
  - merge(link1,link2): Attach linked list2 behind linked list 1
  - search(key): search whether key is in linked list or not

Doubly Linked List

insertatfirst(header, Object o)

1. newnode ← Getnode( )
2. newnode.data ← o
3. newnode.next ← header
4. newnode.prev←NULL
5. header.prev←newnode
6. header ← newnode

Doubly Linked List

insertatlast(header, Object o)

1. newnode ← Getnode( )
2. newnode.data ← o
3. ptr ← header
4. **while** ptr.next ≠ NULL **do**
   a. ptr ← ptr.next
5. ptr.next ← newnode
6. newnode.prev←ptr
7. newnode.next←NULL

Doubly Linked List

insertatany(Object o, Object key): Insert object o at the key position of the linked list.

1. newnode ← Getnode( )
2. newnode.data ← o
3. ptr ← header
4. **while** ptr.data ≠ key **do**
    a. ptr ← ptr.next
5. newnode.next ← ptr.next
6. (ptr.next).prev←newnode
7. ptr.next←newnode
8. newnode.prev←ptr

Doubly Linked List

insertatany(Object o, Object key): Insert object o at the key position of the linked list.

1. ptr ← header
2. **if** ptr = NULL **then**
   a. Print "List Empty"
   b. Exit
3. **while** ptr ≠ NULL **do**
   a. **if** ptr.data = key **then**
      i. newnode ← Getnode( )
      ii. newnode.data ← o
      iii. newnode.next ← ptr.next
      iv. newnode.prev←ptr
      v. ptr.next←newnode
      vi. ptr1←ptr.next
      vii. ptr1.prev←newnode
      viii. Return
4. Print "Data not found"
5. **Return**

Doubly Linked List

deleteatfirst(): Delete object at the starting of the linked list.

1. ptr ← header
2. **if** ptr = NULL **then**
   a. Print “List Empty”
   b. Exit
3. retvalue ← ptr.data
4. header←ptr.next
5. header.prev←NULL
6. Free(ptr)
7. **return** retval

Doubly Linked List

deleteatlast(): Delete object at the last of the linked list.

1. ptr ← header
2. **if** ptr = NULL **then**
   a. Print "List Empty"
   b. Exit
3. **while** ptr.next ≠ NULL **do**
   a. ptr ← ptr.next
4. retval←ptr.data
5. (ptr.prev).next←NULL
6. Free(ptr)
7. **return** retval

Doubly Linked List

innovate achieve lead

deleteatany(key): Delete object at the key position of the linked list.

1. ptr ← header
2. **if** ptr = NULL **then**
   a. Print “List Empty”
   b. Exit
3. **while** ptr.data ≠ Key **do**
   a. ptr ← ptr.next
4. retval←ptr.data
5. (ptr.next).prev←ptr.prev
6. (ptr.prev).next←ptr.next
7. Free(ptr)
8. **return** retval

Doubly Linked List

deleteatany(key): Delete object at the key position of the linked list.

1. ptr ← header
2. **if** ptr = NULL **then**
   a. Print "List Empty"
   b. Exit
3. **if** ptr.data = key **then**
   a. retval ← ptr.data
   b. header ← ptr.next
   c. Free(ptr)
   d. Return retval
4. **while** ptr ≠ NULL **do**
   a. if ptr.data = key **then**
      i. retval ← ptr.data
      ii. ptr1 ← ptr.prev
      iii. ptr2 ← ptr.next
      iv. ptr1.next ← ptr2
      v. ptr2.prev ← ptr1
      vi. Free(ptr)
      vii. Return retval
   b. ptr ← ptr.next
5. Print "Data not found"
6. **Return** NULL

Doubly Linked List

merge(header1,header2,header): Attach list2 behind list1

1. ptr ← header1
2. **while** ptr.next ≠ NULL **do**
   a. ptr ← ptr.next
3. ptr.next←header2
4. header2.prev←ptr
5. header←header1
6. **return** header

Doubly Linked List

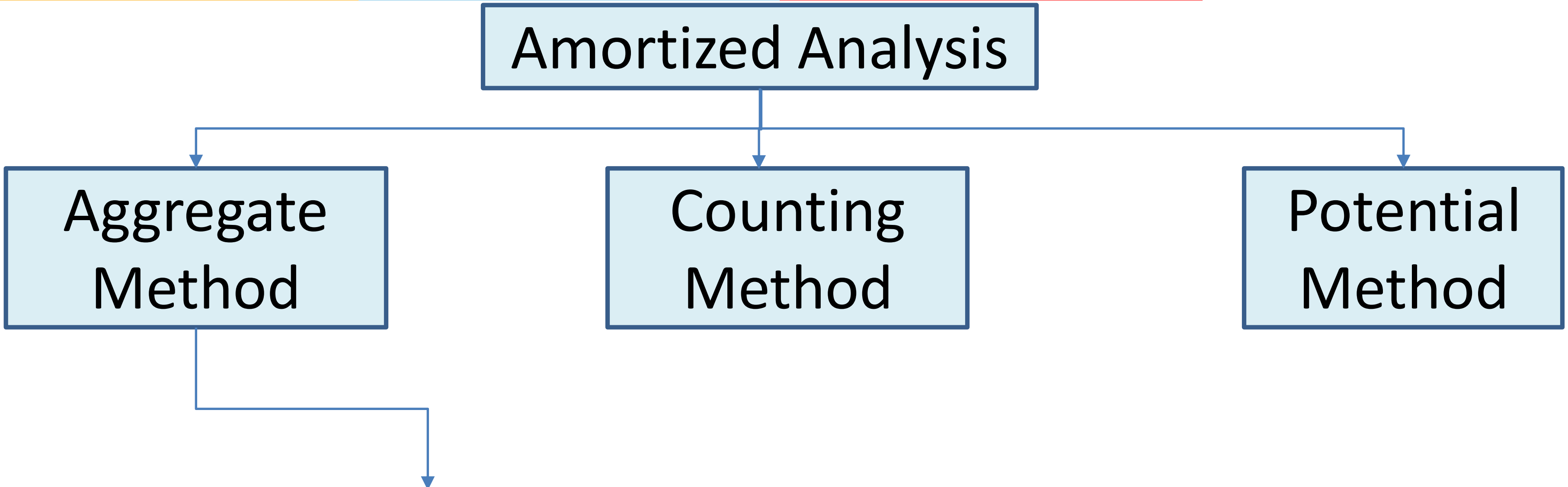
search(key): search whether key is in linked list or not

1. ptr ← header
2. **while** ptr.next ≠ NULL **do**
   a. **if** ptr.data = key **then**
   b. **return** ptr
3. **return** NULL

Amortized Analysis

- Time required to perform a sequence of data-structure operations is averaged over all the operations performed.
- Hypothesis: The average cost of an operation is small, if one averages over a sequence of operations, even though a single operation within the sequence might be expensive
- Amortized Analysis vs. Average Case Analysis
- Amortized analysis does not involve probability
- Amortized analysis guarantees the average performance of each operation in the worst case

Amortized Analysis



T(n): Upper bound on the total cost of a sequence of n operations

Thus, average cost per operation is T(n) / n → every operation has a same amortized cost

Amortized Analysis: aggregate method

Stack Data-Structure:

Push() and Pop() take O(1) time each

→ total cost of a sequence n pus() and pop() operations is n

→ using aggregate method,

- → the amortized cost is T(n) = θ(n) for the sequence of n such operations

Stack Data-Structure:

Consider Multipop(S,k)

While(not Stack-Empty(S) and k!= 0)

    Do Pop(S)

    k ← k -1

Amortized Analysis: aggregate method

What is the runtime of Multipop(S,k) on a stack of s objects?

For each call to Multipop(S,k), the cost is min(s,k), i.e., linear function of this cost

Now, consider a sequence of n operations of Push(), Pop() and Multipop(S,k) on an initially empty stack.

The worst case cost of Multipop() is O(n) since the stack size is at most n.

Thus, the sequence of n operations would cost $O(n^2)$

Although, this analysis is correct, but it is not a tight bound

Amortized Analysis: aggregate method

So, use aggregate analysis to obtain a ***better*** upper bound.

A sequence of n Push(), Pop() and Multipop() operations on an initially empty stack will cost O(n)

Why?

Each object can be popped at most once for each time it is pushed into the stack

The number of Pop(), including the ones within Multipop(), can be called on a non-empty stack is at most the number of Push() calls.

The number of Push() calls can be at most n because it is the maximum size of the stack

Amortized Analysis: aggregate method

So, for any value of n, any sequence of n push(), pop() and multipop() operations takes a total of O(n) time.

Thus, average cost of an operations in this sequence is O(n)/n = O(1).

References

1. Algorithms Design: Foundations, Analysis and Internet Examples Michael T. Goodrich, Roberto Tamassia, 2006, Wiley (Students Edition)
2. Data Structures, Algorithms and Applications in C++, Sartaj Sahni, Second Ed, 2005, Universities Press
3. Introduction to Algorithms, TH Cormen, CE Leiserson, RL Rivest, C Stein, Third Ed, 2009, PHI

Any Question!!

Thank you!!

BITS Pilani
Hyderabad Campus