REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

Université de M'sila
Faculté des Mathématiques et de l'Informatique
Département d'informatique

جامعة المسيلة
كلية الرياضيات والإعلام الآلي
قسم الإعلام الآلي

Chapter 3: Part 3
Pointers and Linked Lists

Algorithmics and data structure 2

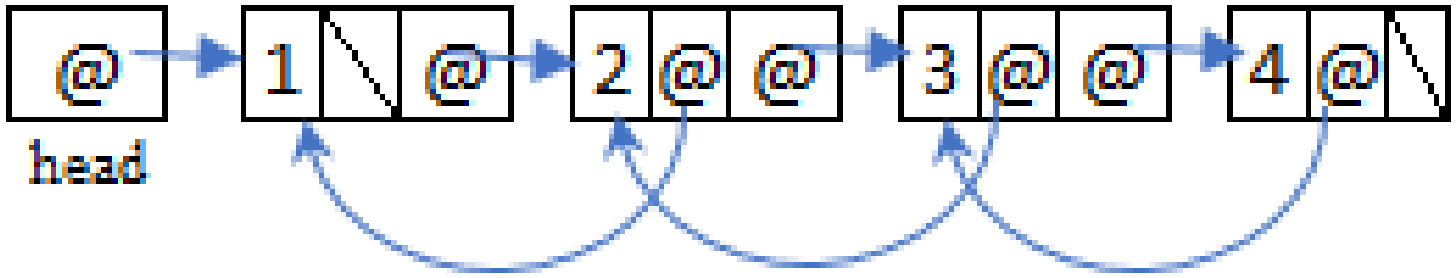
Presented by : Dr. Benazi Makhlouf
Academic year : 2023/2024

Content of chapter 03 part 3:

7. Doubly linked list
8. Special linked lists
    1. Stacks
    2. Queues

7. Doubly linked list

In addition to the data and the pointer that points to the next element, a doubly linked list contains another pointer, usually called "prev", that points to the previous element. This pointer makes it easy to navigate the list in both directions and thus simplifies the process of deleting or inserting an element before the selected one.



declaration

```
typedef struct Node {
   int data ;
   struct Node*next, *prev;
} Node;
```

"next" is a pointer that contains the address of the next element,

"prev" is a pointer that contains the address of the previous element.

The "prev" of the first element can be used to refer to the last element in the list, speeding up the process of accessing the last element for addition or deletion.

Add an element at the beginning(head)

```
e->next = h ; //
```
Change next from “e” so that it points to the first

```
e->prev = NULL;

   if(h != NULL) //
```
prev of the first element, if it exists, points to the new .

```
    h->prev = e;
h =e; //
```
the head of the list points to the new element

Delete the element from the beginning(the head)

```
t = h ; //
```
Store the address of the element to delete

```
h =t->next; //
```
Bind with the second element

```
   if(h != NULL)

    h->prev = NULL;
   delete t; //
```
Empty reserved memory

Add an element at the end

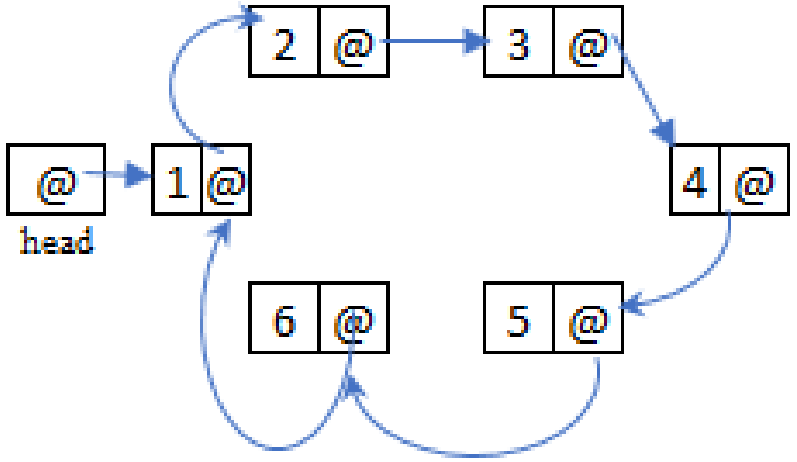
```
e->next = NULL; //  because it will be the last
 if(h == NULL) {
    e->prev = NULL;
    h = e;
}
 else {
    t= h ;
    while(t-> next != NULL) //  the last element is searched
      t= t->next;
    e-> prev = t; //connect the new one with the last one
    t->next=e;
}
```

Remove an item from last

```
t = h ;

while(t->next != NULL) //  the last element is searched

   t= t->next;
p=t->prev; // get the address of the before last

p ->next=NULL; // become the last

delete t;
```

8. Special linked lists

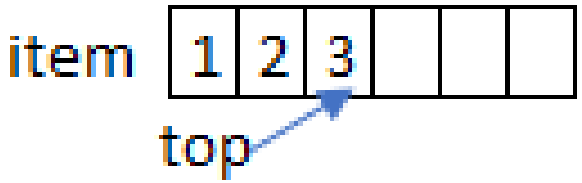
In addition to single and double linked lists, there are circular single and double linked lists. A circular list is similar to a normal linked list, with the distinction that the last element does not have a NULL reference but instead points to the first element of the list.



8.1. Stacks:

Stack:

- An abstract data structure consisting of a set of records of the same type.
- The two fundamental operations are 'push'(to add items to the stack) and 'pop'(to remove items from the stack).
- Operations take place at a single end of the group, referred to as the top.
- The data structure follows the LIFO(Last In, First Out) principle.
- The last item added is the first to be removed, and the output order is the reverse of the input order.
- Can be implemented using arrays or linked lists.



Examples:

- Web browser history
- The list of operations in Word to be undone as items.

8.1.1. Using arrays

1. **Declaration** : A structure is created that contains a dynamically allocated item table, a 'top' field representing the position for addition or deletion, and a 'capacity' variable indicating the size."

```
typedef struct Stack {
   int *item;
   int top, capacity ;
} Stack;
```

2. **init()**: The array is created and top is assigned the value 0 to indicate that the stack is empty.

```
Stack init(int size) {
   Stack s;
   s.top = 0;
   s.capacity = size ;
   s.item =new int[size];
   return s;
}
```

3. **isEmpty()**:The stack is empty if the value of ‘top’ is 0.

```
bool isEmpty(Stack s){
    return s.top ==0;
}
```

4. isFull(): If the array is full, ‘top’ equals ‘capacity’ .

```
bool isFull(Stack s){
  return s.top == s.capacity;
}
```

5. **Pop**(): The Pop function decrements 'top' and returns the last element it points to.

```
int Pop(Stack &s){
  int x;
  if(isEmpty(s)){
    printf("error: Stack is empty");
    return -1;
  }
  s.top--;
  x= s.item[s.top];
  return x;
}
```

6. **Push**(): The Push function adds the element 'x' to the table and increments the 'top' pointer by 1. It is important to ensure that the stack(table) is not full before performing the operation.

```
void Push(Stack &s, int x){
  if(isFull(s)){
    printf("error: Stack is full");
    return;
  }
  s.item[s.top]=x;
  s.top++;
}
```

8.1.2. Using linked lists:

To simulate a stack using lists, the addition and removal must be done on the same side(at the beginning or at the end).

1. **Push()**: The push function is the same as the add_head function

```
void Push(List &l , int x) {
  List e = new Node;
  e->data = x;
  e->next = l ;
  l =e;
}
```

2. **Pop()**: The pop function is the same as the delete_head function except that the pop function returns the element that was deleted. So before deleting the first element t, we save t->data in x, then delete it and return the value of x.

```
int Pop(List &l) {
  List t; int x;
  if(l==NULL){
    printf("error: Stack is empty");
    return -1;
  }
  t = l;
  l =t->next;
  x=t->data;
  delete t;
  return x;
}
```

8.2. The Queue

Queue:

- An abstract data structure for storing records of the same type.
- Supports two fundamental operations:
  - Adding a new element(enQueue or enfil).
  - Deleting an element(deQueue or scroll).
- Follows the FIFO(First In, First Out) property.
- The first element added is the first element to be deleted.
- Output order matches the input order.
- Can be implemented using arrays or linked lists.

Example : list of events, list of files sent to the printer...

8.2.1. Using linked lists:

To simulate a queue using lists, you perform additions and removals at two different ends of the list. Specifically:

- Add new elements at the end of the list.
- Delete elements from the beginning of the list.

Alternatively, this approach can be reversed by:

- Adding items at the beginning of the list.
- Removing items from the end of the list.

The list structure allows for quick and efficient insertions and deletions, without requiring costly element moves, as is the case with array implementations.

1. **declation** : We create a structure that contains two fields, the first refers to the first element of the list and the second to the last element of the list.

```
typedef struct {
  struct Node *first, *last;
  int size;
} Queue;
```

initQ() initializes the queue by assigning **NULL** to the first and last fields of the structure.

```
Queue initQ(){
 Queue Q;
 Q.first =NULL;
 Q.last =NULL;
 Q.size =0;
 return Q
}
```

```
bool isEmpty(Queue Q){
    return Q.size ==0;
}
```

enQueue():is the same as “ append_end ”

```
void enQueue(Queue &Q, int x) {
  Node *e = new Node;
  e->data = x;
  e->next = NULL;
  if(isEmpty(Q))
    Q.first =e;
  else
    Q.last ->next =e;
  Q.last =e;
  Q.size++;
}
```

deQueue():is the same as “ delete_head”

```
int deQueue(Queue &Q) {
  Node*t; int x;
  if(isEmpty(Q)){
    printf("error: is empty");
    rturn -1;
}
  t = Q.first ;
  x = t ->data;
  Q.first = t->next;
  delete t;
  Q.size--;
  if(Q.size ==0) Q.last =NULL;
  return x;
}
```

8.2.2. Using arrays

1. **Declaration** : Creating a structure that includes a dynamically allocated table of elements in memory, a 'start' location for additions, an 'end' location for deletions, and 'capacity,' which indicates the maximum number of elements that can be added to the table.

```
typedef struct {
   int *item;
   int start, end, capacity;
} Queue;
```

2. **init()**: Creates the table and sets start and end to -1 to indicate that the queue is empty. If the creation fails, the function returns false.

```
Queue init(int size) {
   Queue Q;
   Q.start = -1;
   Q.end = -1;
   Q.capacity = size;
   Q.item = new int[size];
   return Q;
}
```

3. **isEmpty()**: checking if the queue is empty by comparing if start =-1

```
bool isEmpty(Queue Q){
    return Q.start ==-1 && Q.end ==-1;
}
```

4. **isFull()**: Checks if the queue is full by comparing whether the value of(start + 1) modulo the capacity is the same as the value of end using the mod "%" operation to handle the case where the queue wraps around to the start.

```
bool isFull(Queue Q){
 return(Q.start+1) % Q.capacity == Q.end;
}
```

5. **enQueue()**: Checks if the queue is not full, then adds 1 to start to reference the first empty element and adds x to the table.

```
void enQueue(Queue &Q,int x){
  if(isFull(Q)){
    printf("error: Queue is full");
    return;
}
  if(isEmpty(Q))  Q.end =0;
  Q.start=(Q.start+1) % Q.capacity;
  Q.item[Q.start]=x;
}
```

6. **deQueue**(): returns the first element of the table pointed to by end, If the queue is empty, it informed the user. If the queue becomes empty, put -1 in start otherwise add 1 to end.

```
int deQueue(Queue &Q) {
  if(isEmpty(Q)) {
    printf("error: Queue is empty");
    return -1;
  }
  int x = Q.item[Q.end];
  if(Q.start == Q.end)
    Q.start = Q.end = -1;
  else
    Q.end =(Q.end + 1) % Q.capacity;
  return x;
}
```

End of Chapter 03