Data Structures and Algorithms

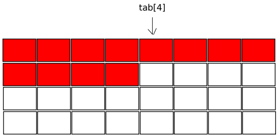
Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

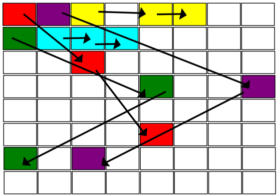
Static vs. **Dynamic**

- Static DS (*i.e.*, arrays) are "memory-friendly" and allow *constant-time* access to any element.

Downside: erasure of an internal element...



- Dynamic DS (*i.e.*, **lists**) allow easy addition/removal of elements in the collection.

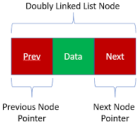


(Singly) Linked Lists

- Elements may not be consecutive in the memory.
- Each element is allocated separately in an individual sub-structure, sometimes called a *node*.
  - The information zone of a node is the variable field containing the data unit
  - The link zone of a node contains a pointer toward the next node in the list.
- The unique entry point to the DS is the first node (**head** of the list). It is the only node w/o a predecessor. The **tail** is the only node w/o a successor.

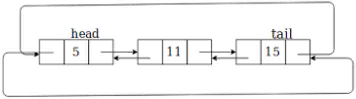
Variations

- A **Doubly Linked List** augments every node with a pointer to the previous node (if it exists).



- A (Singly Linked) **Circular List** modifies the link zone of the tail node so that it points to the head.

- A **Circular Doubly Linked List** modifies the link zone of both the head and the tail, so that they point to each other.



Implementation

- Definition of a node data structure:

```
struct node {
  //information zone
   int info;
  //link zone
  node *next;
  node *pred; //only if doubly linked
}
typedef node *LinkedList; //pointer to the head of the list
```

- For a doubly linked list, we may also want to have access to the tail

```
struct DoublyLinkedList {
   node *head, *tail;
}
```

Classical operations on lists

```
//Emptiness Test. Complexity: O(1)
bool empty(const LinkedList& L) { return (L==nullptr); }

//Search by index. Complexity: O(i).
int get(const LinkedList& L, int i) {
   node* n = L;
   for(int j = 1; j <= i; j++)
      n = n->next; //I am too lazy to check whether it exists...
   return n->info;
}

//Special Case. Complexity O(1).
int head(const LinkedList& L) { return get(L,0); }
```

Number of elements

Function size()

- In $\mathcal{O}(n)$-time (easy)

```
int size(const LinkedList& L) {
   int s(0);
   node *tmp = L;
   while(tmp) {
      s++; tmp = tmp -> next;
   }
   return s;
}
```

Number of elements

Function size()

- In $\mathcal{O}(n)$-time (easy)

```
int size(const LinkedList& L) {
   int s(0);
   node *tmp = L;
   while(tmp) {
      s++; tmp = tmp -> next;
   }
   return s;
}
```

- In $\mathcal{O}(1)$-time: Store the size in an additional variable.

$\longrightarrow$ to be updated after each modification!

```
struct LinkedList {
   node *head = nullptr, *tail = nullptr;
   int size = 0;
}
```

Classical operations revisited

```
//Complexity: O(1)
int size(const LinkedList& L) { return L.size; }

//Complexity: O(1)
bool empty(const LinkedList& L) { return (L.size==0); }

//Complexity: O(i).
int get(const LinkedList& L, int i) {
  //if(i >= 0 && i < L.size) ...
  node* n = L.head;
  for(int j = 1; j <= i; j++)
    n = n->next;
  return n->info;
}

//Complexity O(1).
int head(const LinkedList& L) { return get(L,0); }
int tail(const LinkedList& L) { return (L.tail)->info; }
```

Modification of the list

Insertion

```
//Assumption: 0 ≤ i ≤ L.size. Complexity: O(i)
void add(LinkedList& L, int e, int i = 0) {
   L.size++;
   node *n = new nod;
   n -> info = e;
   if(i==0) { //new head
      n->next = L.head;
      L.head = n;
   } else {
      node *p = L.head; //pred.
      for(int j = 1; j < i; j++)
         p = p->next;
      n->next = p->next;
      p->next = n;
   }
   if(!n->next) { L.tail = n; } // new tail
}
```

Complements on Insertion

- For a doubly linked list, one also needs to update the pointer to previous elements.

```
...
if(i==0) {
  ...
  n->pred = nullptr;
} else {
  ...
  n->pred = p;
}
if(!n->next) { ...}
else { n->next->pred = n; }
```

- For a circular list, no pointer can be null (we “circle back”)

```
...
if(i==0) {
  if(L.size == 0) { //empty list
     L.head = L.tail = n;
     n->next = n;
  } else {
     ...
     (L.tail)->next = n;
} else {
  ...
  if(n->next == L.head) {
     L.tail = n; }
}
```

Modification of a list

Removal

```
//Assumption: 0 ≤ i < L.size. Complexity: O(i)
void remove(LinkedList& L, int i = 0) {
    L.size--;
    node *n; //node to be removed
    if(i==0) { //new head
        n = L.head;
        if(L.size == 0) { L.head = L.tail = nullptr; } //emptied list
        else { L.head = n->next; }
    } else {
        node *p = L.head; //pred.
        for(int j = 1; j < i; j++)
            p = p->next;
        n = p->next;
        p->next = n->next;
        if(!p->next) { L.tail = p; } // new tail
    }
    delete n;
}
```

Complements on Removal

- For a doubly linked list, one also needs to update the pointer to previous elements.

```
...
if(i==0){
  ...
  else{
    L.head = n->next;
    L.head->pred = nullptr;
}} else {
    ...
    if(!p->next) {L.tail = p;}
    else {p->next->pred = p;}
}
...
```

- For a circular list, no pointer can be null (we "circle back")

```
...
if(i==0){
  ...
  else{
    L.head = n->next;
    if(L.size==1)
      L.head -> next = L.head;
}} else {
  ...
  if(p->next == L.head)
    L.tail = p;
}
...
```

The special case of the **tail**

- Our implementations require $\mathcal{O}(n)$ for adding/removing a node at the tail of the list.
  - For singly linked lists this is unavoidable: need to find the new tail.
  - However, for **doubly** linked list, this can be done in $\mathcal{O}(1)$

Example:

```
void addLast(DoublyLinkedList& L, int e) {
   node *n = new nod;
   n->info = e; n->next = nullptr;
   if(empty(L)) {
      L.head = n;
   } else {
      (L.tail)->next = n;
   }
   n->pred = L.tail;
   L.tail = n;
}
```

Range Queries on Lists

- Mo's algorithm can still be applied

  $\longrightarrow$ we need to keep a pointer to first and last node of each block.

- However, Binary Search cannot be applied to speed up Searching in ordered lists.

  $\longrightarrow$ access to the median already requires $\mathcal{O}(n)$.

Questions