Linked Lists

Linked List Basics

- Linked lists and arrays are similar since they both store collections of data.
- *Arrays* allocate memory for all elements at the same time and in one block of memory.
- *Linked lists* allocate memory for each element separately and only when necessary.

Disadvantages of Arrays

1. **The size of the array is fixed**.
   - In case of **dynamically resizing** the array from size S to 2S, we need 3S units of available memory.
   - Programmers allocate arrays which seem **"large enough**" This strategy has two disadvantages: (a) most of the time there are just 20% or 30% elements in the array and 70% of the space in the array really is wasted. (b) If the program ever needs to process more than the declared size, the code breaks.
2. **Inserting (and deleting)** elements into the middle of the array is potentially expensive because existing elements need to be shifted over to make room

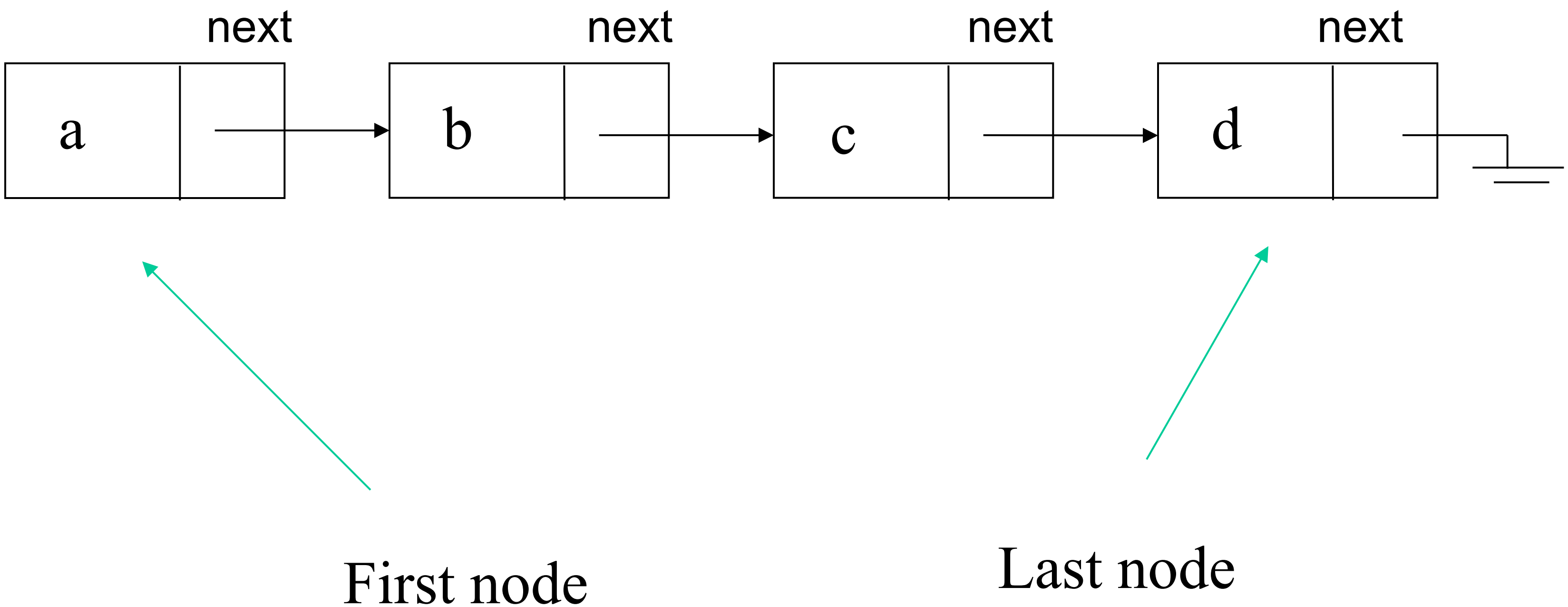
Linked lists

- Linked lists are appropriate when the number of data elements is unpredictable.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- Each node does not necessarily follow the previous one physically in the memory.
- Linked lists can be maintained in sorted order by inserting or deleting an element at the proper point in the list.
- However, you can’t directly jump to any i^{th} element in the linked list. You need to apply sequential search.
- Arrays, on the other hand, allow direct access to any i^{th} element. No need for sequential access.

Pointers and Linked Lists

- To represent linked lists we will use pointers.
- Pointer: also called as link or reference, is a variable that gives location of some other variable.
- Linked List: For every element in the list, we put a pointer into the element giving the location of the next element in the list.

Single Directional Linked Lists (Singly Linked Lists)

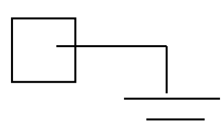


Empty List

- Empty Linked list is a single pointer having the value of NULL.

```
first = NULL;
```

first



Implementation of Linked Lists

- Using Arrays

```
typedef struct linkedList{
    char data[5];
    int next;
}SIMPLE_LIST1;

//This list has at most 100 elements
SIMPLE_LIST1 L[100];
int first = -1; //Index of the first element
int last = -1; //Index of the last element
```

Implementation of Linked Lists

- Using Pointers

```
typedef struct linkedList{
   char data[5];
   struct linkedList *next;
}SIMPLE_LIST1;

//This list has any number of elements
//Pointers to the first and last elements
// Initially, the list is empty
SIMPLE_LIST1 *first = NULL;
SIMPLE_LIST1 *last = NULL;
```

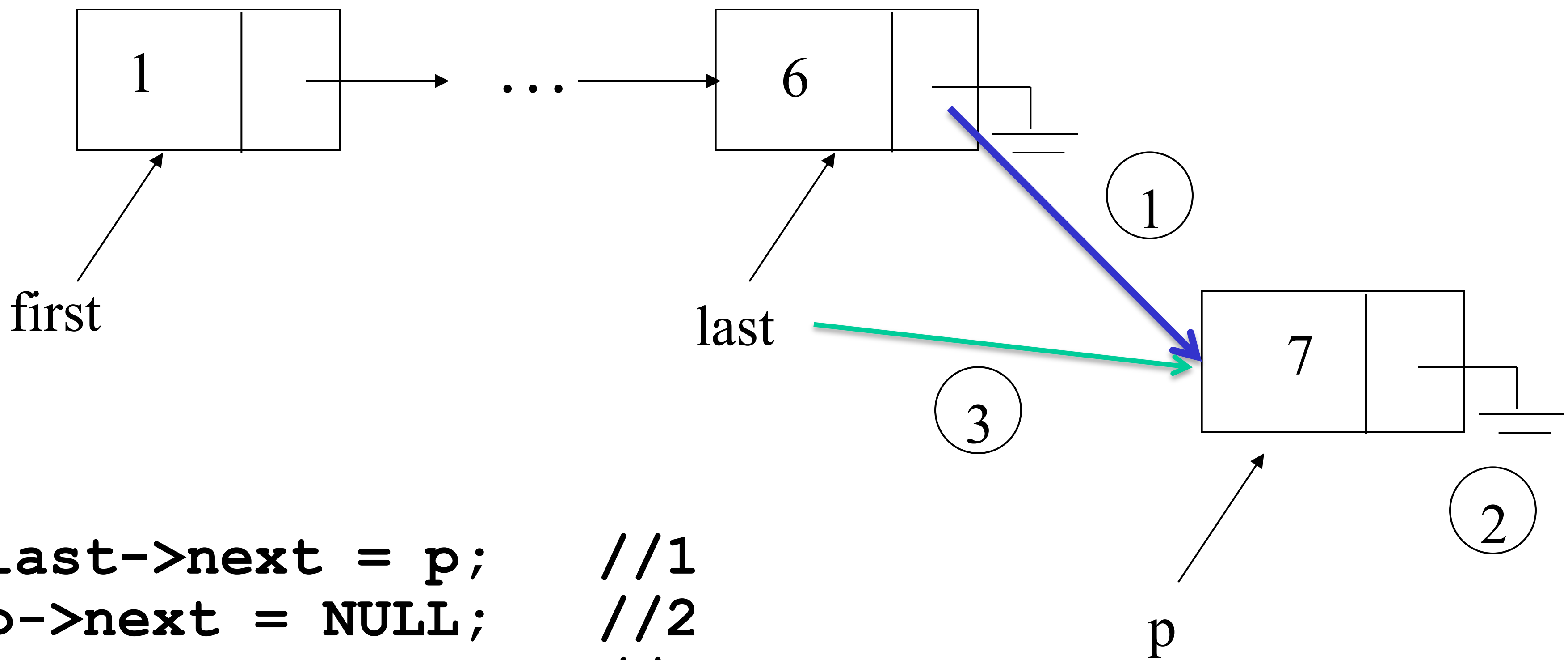
Basic Linked List Operations

- Define the List
- Insert a node
- List Traversal
- Searching a node
- Delete a node

Definition of the List

```
typedef struct linkedList{
    int info;
    char message[100];
    struct linkedList *next;
}SIMPLE_LIST;

// Initially, the list is empty
SIMPLE_LIST *first = NULL;
SIMPLE_LIST *last = NULL;
```

Insert a new element at the end




```
last->next = p;    //1
p->next = NULL;    //2
last = p;          //3
```

What happens if the list is empty?

```
first = p;
last = p;
p->next = NULL;
```

Insertion function

```
int insert(SIMPLE_LIST *p){
   if (first != NULL){
      last->next = p;
      p->next = NULL;
      last = p;
   }
   else {
      first = p;
      last = p;
      p->next = NULL;
   }
   return 0;
}
```

Traversing a linked list

```
int displayList(){
   SIMPLE_LIST *p
   p = first;
   if (p == NULL){
      printf("List is empty\n");
      return -1;
   }
   while (p != NULL){
      printf(" %d %s \n",
              p->info, p->message );
      p = p->next;
   }
   return 0;
}
```

Searching a node in a linked list

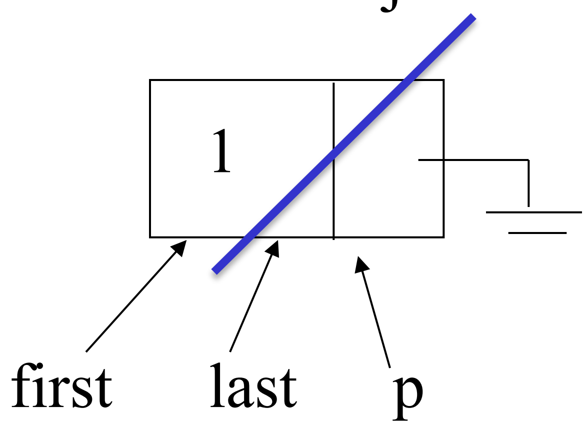
```
// Use sequential search

// Search until target is found or we reach
// the end of list

SIMPLE_LIST *search(int key){
   SIMPLE_LIST *p;

   p = first;
   while (p){
      if (p->info == key)
         return p;
      p = p->next;
   }
   return NULL;
}
```

Deletion from a linked list

Search for the element to be deleted.

if it is the first element

if the list has just one element:

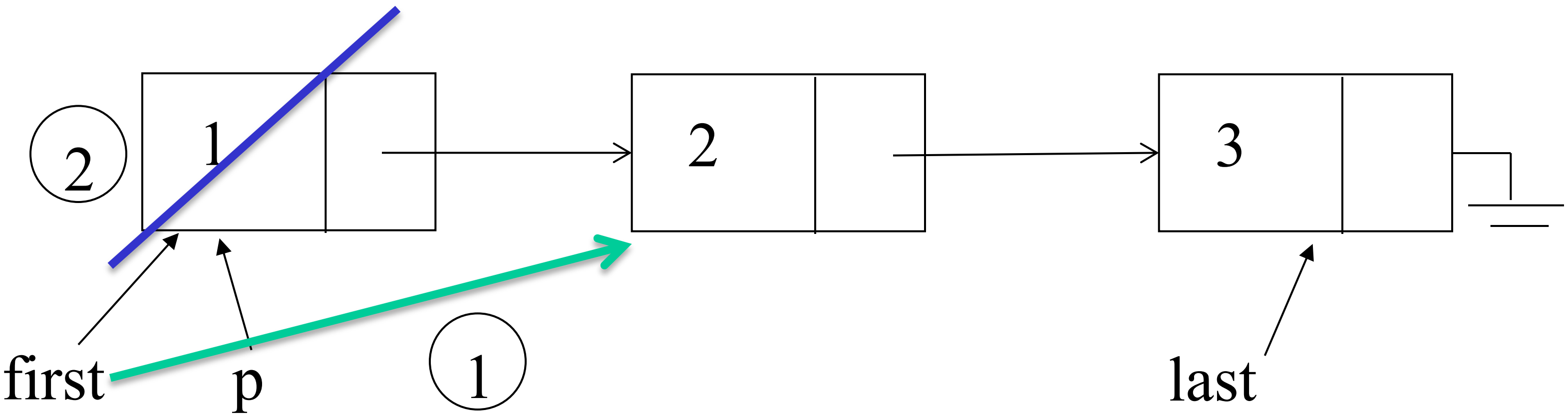



```
// Empty the list
first = NULL;
last = NULL;
free(p);
```

Deletion from a linked list

if it is the first element

if the list has more than one element:

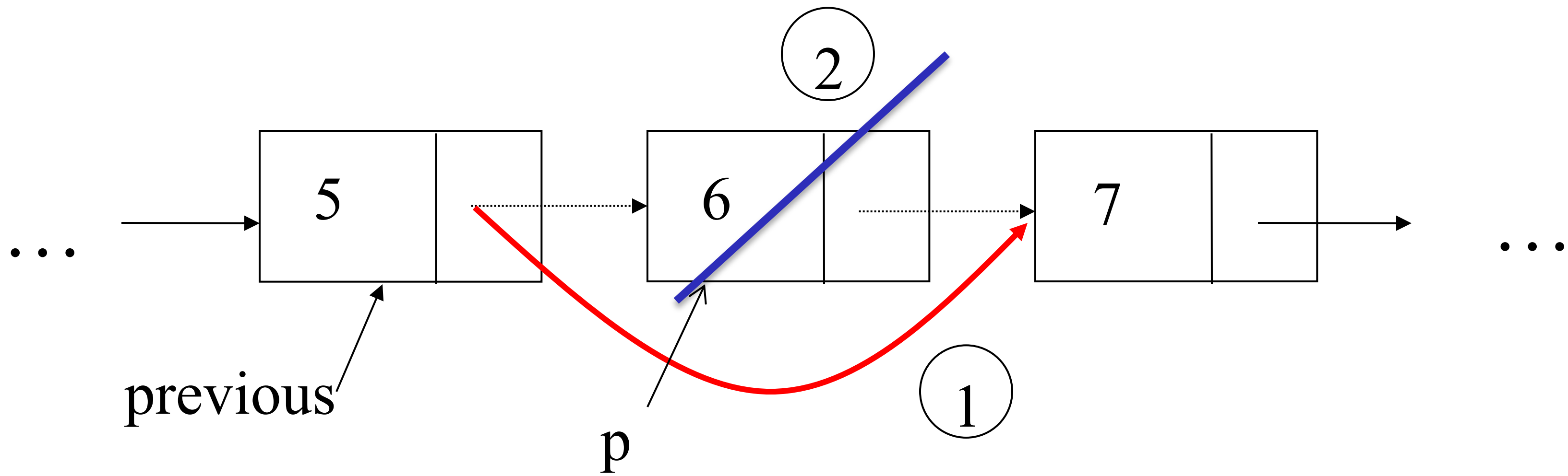



```
first = p->next;    //1
free(p);            //2
```

Deletion from a linked list

if it is in the middle or last

find the previous element, and update the pointers




```
previous->next = p->next;  //1
free(p);                   //2
if (p == last)
    last = previous;
```

Deletion function

```
SIMPLE_LIST *delete(int key){
   SIMPLE_LIST *p, *previous;
   p = first;
   previous = NULL;

// search for the element to be deleted

   while (p){
      if (key == p->info)
         break;
      previous = p;
      p = p->next;
   }
```

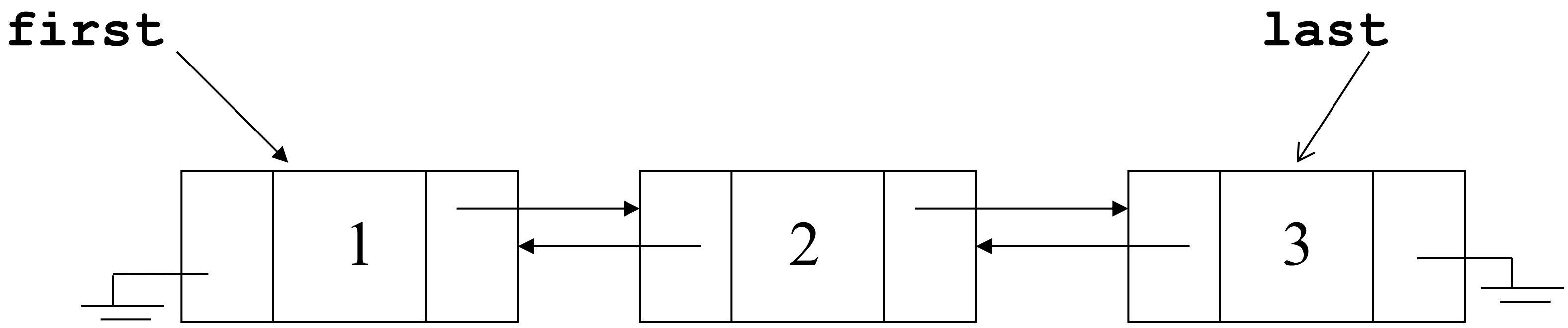
Deletion function (cont.)

```
if (p != NULL){ //if found
   if (previous == NULL){
   // if first element will be deleted
      if (first == last){
      // if list has one element
         first = NULL;
         last = NULL;
      }
      else {
          first = first->next;
      }
```

Deletion function (cont.)

```
        else{
         //delete from middle or last
         previous->next = p->next;
         if (previous->next == NULL){
            //last element is deleted
            last = previous;
         }
      }
      free(p);
      return(p);
   }
   else     //not found
      return NULL;
}
```

Doubly Linked Lists



Advantages:

- Convenient to traverse the list backwards.
- Simplifies insertion and deletion because you no longer have to refer to the previous node.

Disadvantage:

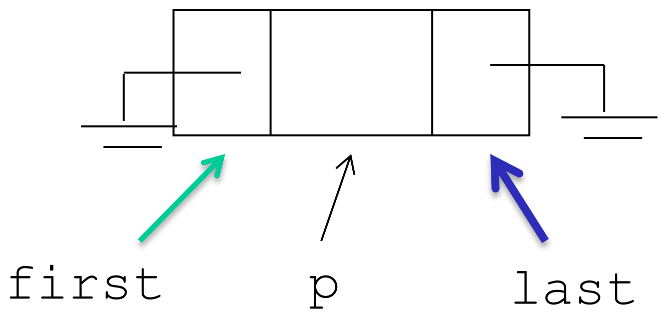
- Increase in space requirements.

Definition of the list

```
typedef struct doubly_list{
    int info;
    char message[100];
    struct doubly_list *previous;
    struct doubly_list *next;
}DLIST;
DLIST *first = NULL;
DLIST *last = NULL;
```

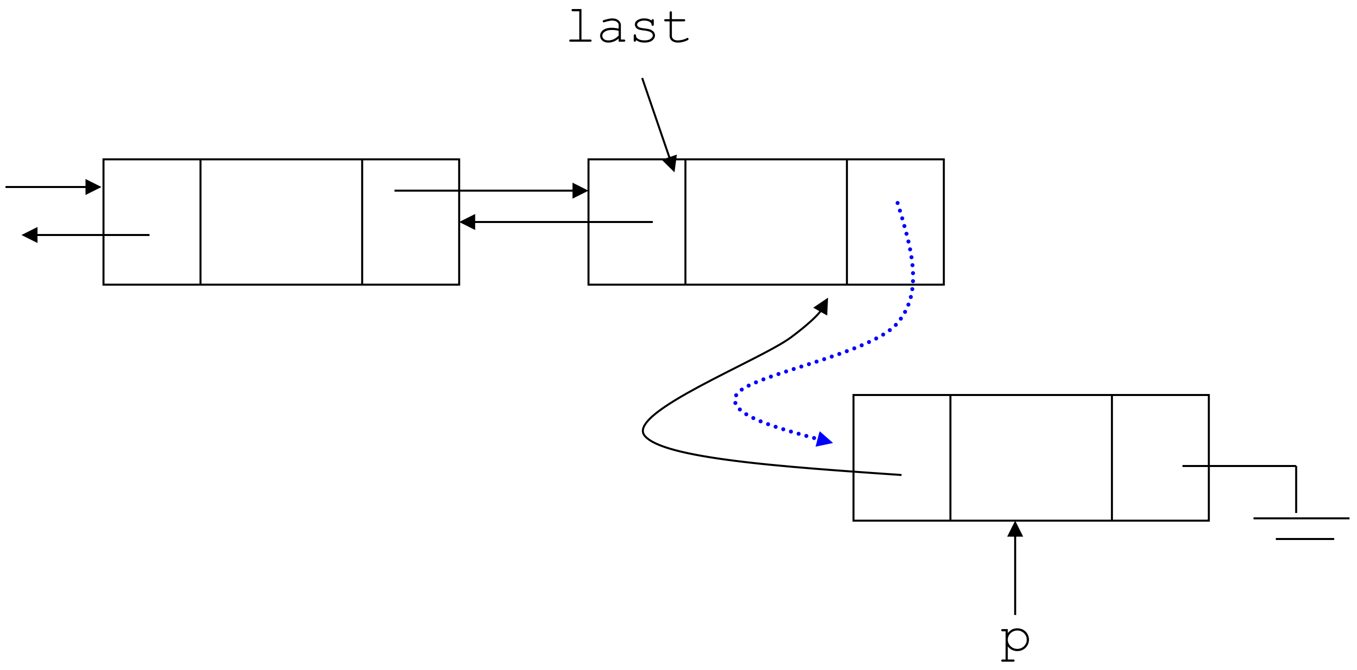
Insertion

if the list is empty




```
first = p;
last = p;
```

Insertion (cont.)




```
// insert at the end
last->next = p;
p->previous = last;
p->next = NULL;
last = p;
```

Insertion function

```
int insert(DLIST *p){
   if (first != NULL){  // if list is not empty
      last->next = p;
      p->previous = last;
      p->next = NULL;
      last = p;
   }
   else {   // if list is empty
      first = p;
      last = p;
      first->previous = NULL;
      last->next = NULL;
   }
   return 0;
}
```

Display the list on screen

```
int display(){
   DLIST *p;
   p = first;
   if (p == NULL){
      printf("List is empty\n");
      return -1;
   }
   while (p){
      printf("%d %s\n", p->info, p->message);
      p = p->next;
   }
   return 0;
}
```

Search an element from the list

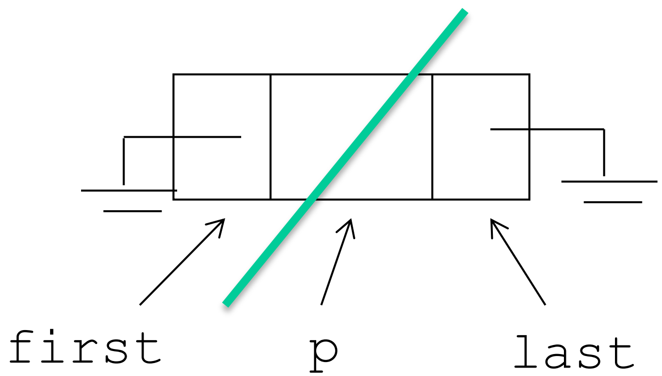
```
DLIST *search(int key){
   DLIST *p;
   p = first;
   while (p){
      if (key == p->info)
         return p;
      p = p->next;
   }
   return NULL;
}
```

Deleting an element

- Search for the element to be deleted.
- If it is found
  - If it is the first element
    - If the list has only one element
      - Empty the list
    - Else
      - Delete the first element, and update first
  - Else
    - If it is in the middle, delete from middle
    - Else, delete from last element

Deleting the first element

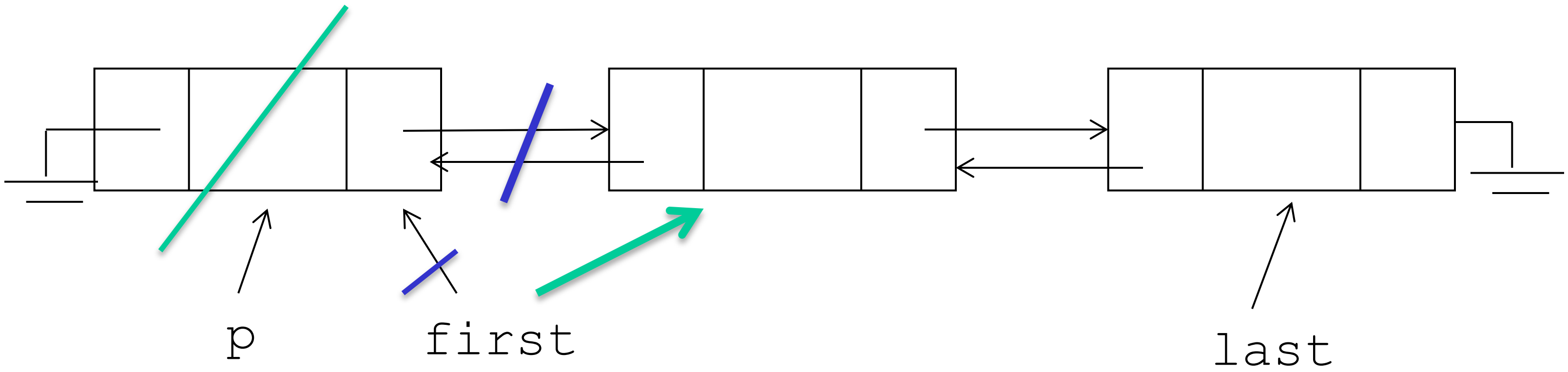
if the list has only one element, empty the list




```
first = NULL;
last = NULL;
free(p);
```

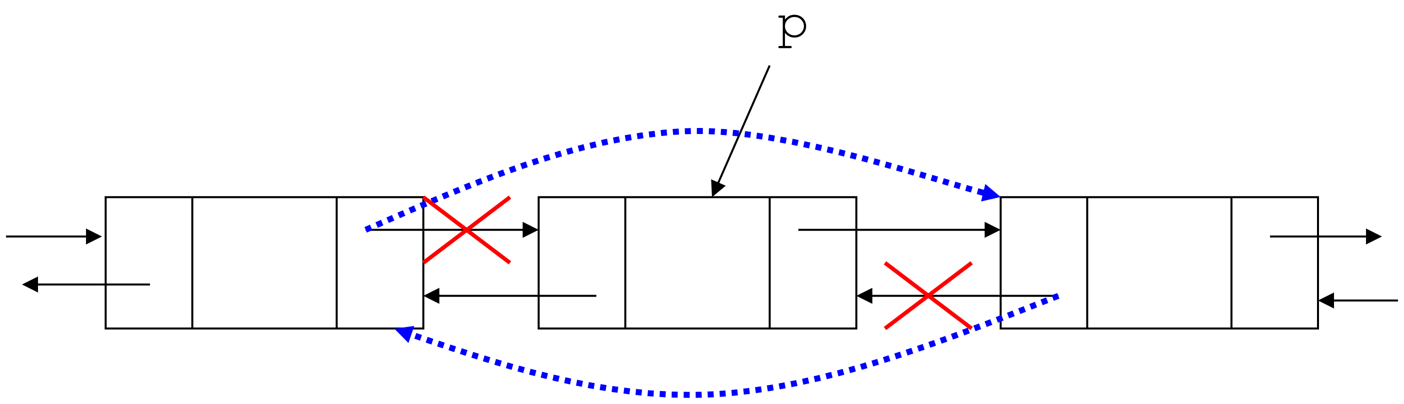
Deleting the first element

if the list has more than one element, update first



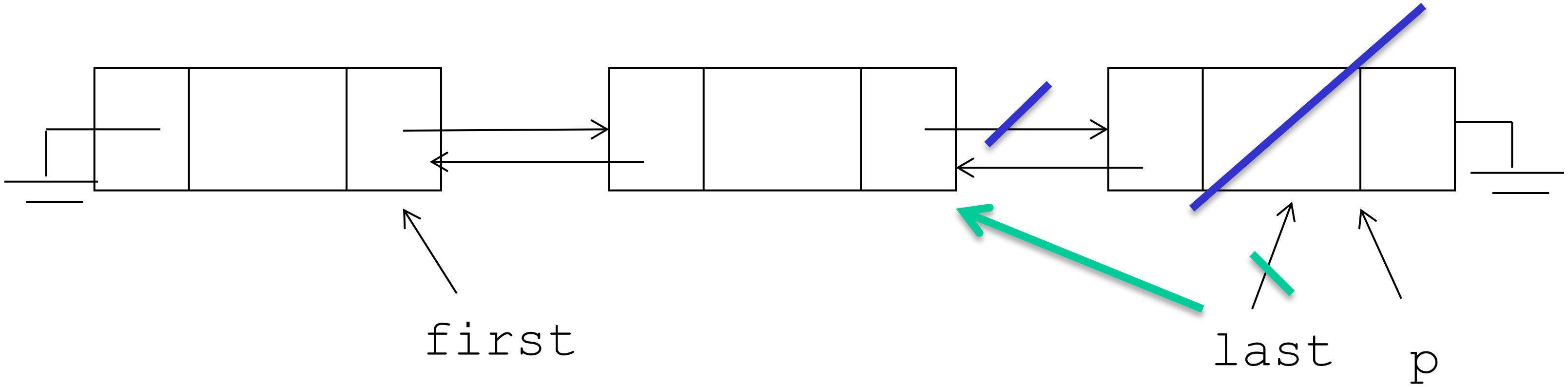

```
first = p->next;
first->previous = NULL;
free(p);
```

Deleting from middle




```
p->previous->next = p->next;
p->next->previous = p->previous;
free(p);
```

Deleting the last element




```
last = p->previous;
last->next = NULL;
free(p);
```

Deletion function

```
DLIST *delete(int key){
   DLIST *p;
   p = search(key);
   if (p==NULL){
      printf("The element to be deleted is not
              in the list\n");
      return NULL;
   }
   if (p == first){  //Delete the first element
      if (first == last){  // list has 1 element
         first = NULL;
         last = NULL;
      }
```

Deletion function (cont.)

```
        else {  // list has more than one element
            first = p->next;
            first->previous = NULL;
        }
    }
    else{
        if (p == last){   //Delete from last
            last = p->previous;
            last->next = NULL;
        }
        else { //Delete from middle
            p->previous->next = p->next;
            p->next->previous = p->previous;
        }
    }
    free(p);
    return p;
}
```

Saving and Restoring a Linked List by Using a File

- Use an external file to preserve the list
- Do not write pointers to a file, only data
- For each element in the list
  - Copy the element into a file

- Recreate the list from the file by placing each item at the end of the list
- For each element in the file
  - Insert the element at the end of the list

Saving a list in a file

```
int store(){
   FILE *fp;
   DLIST *p;
   // open the file
   if ((fp=fopen("list.txt","w"))==NULL){
      printf("File cannot be opened, disk is full\n");
      return -1;
   }
   p = first;
   while (p){
      fwrite(p, sizeof(DLIST)-2*sizeof(p), 1, fp);
      p = p->next;
   }
   printf("List was stored\n");
   fclose(fp);
   return 0;
}
```

Comparing Array-Based and Pointer-Based Implementations

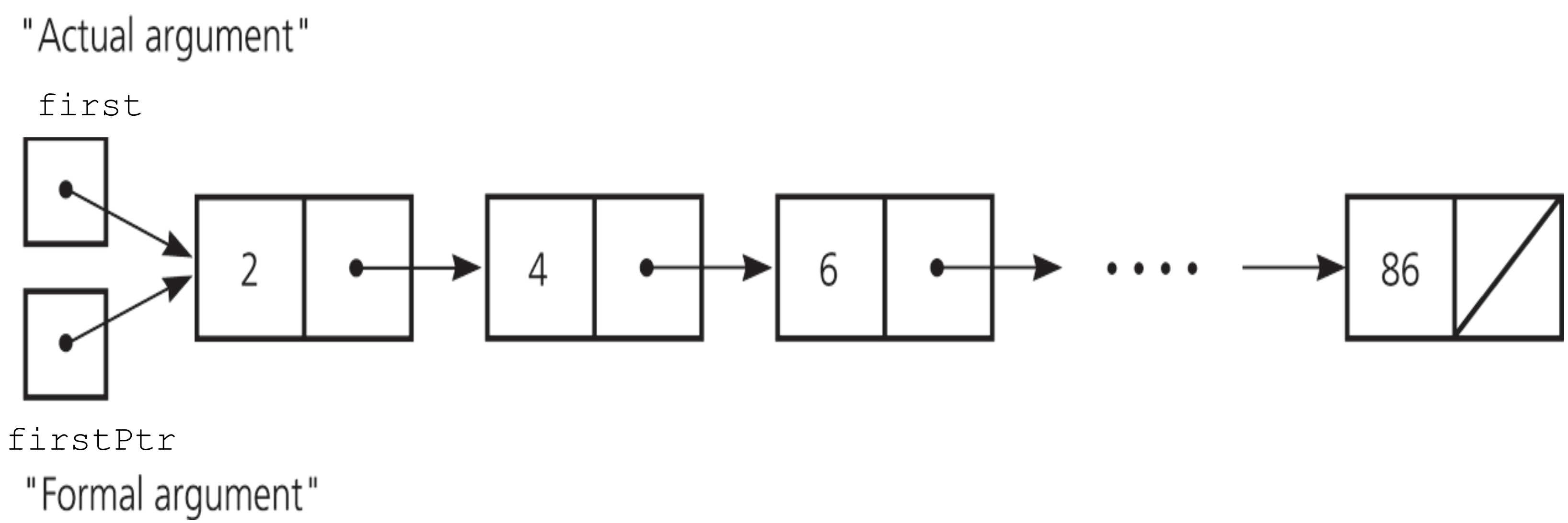
- Size
  - Increasing the size of a resizable array can waste storage and time
- Storage requirements
  - Array-based implementations require less memory than a pointer-based ones

Comparing Array-Based and Pointer-Based Implementations

- Access time
  - Array-based: constant access time
  - Pointer-based: the time to access the i^{th} node depends on i
- Insertion and deletions
  - Array-based: require shifting of data
  - Pointer-based: require a list traversal

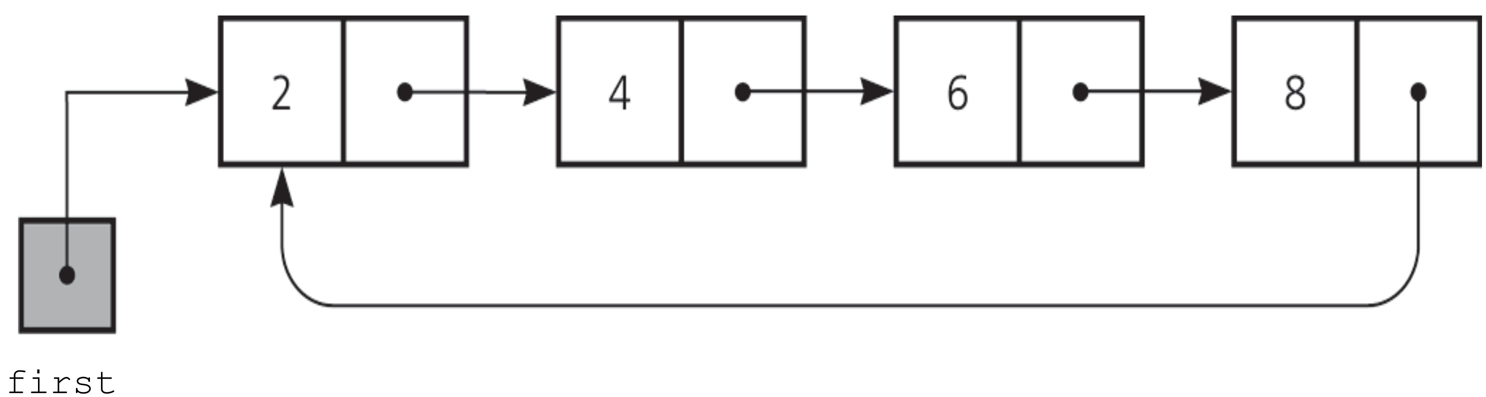
Passing a Linked List to a Function

- A function with access to a linked list's `first` pointer has access to the entire list
- Pass the `first` pointer to a function as a reference argument



Circular Linked Lists

- Last node references the first node
- Every node has a successor
- No node in a circular linked list contains *NULL*

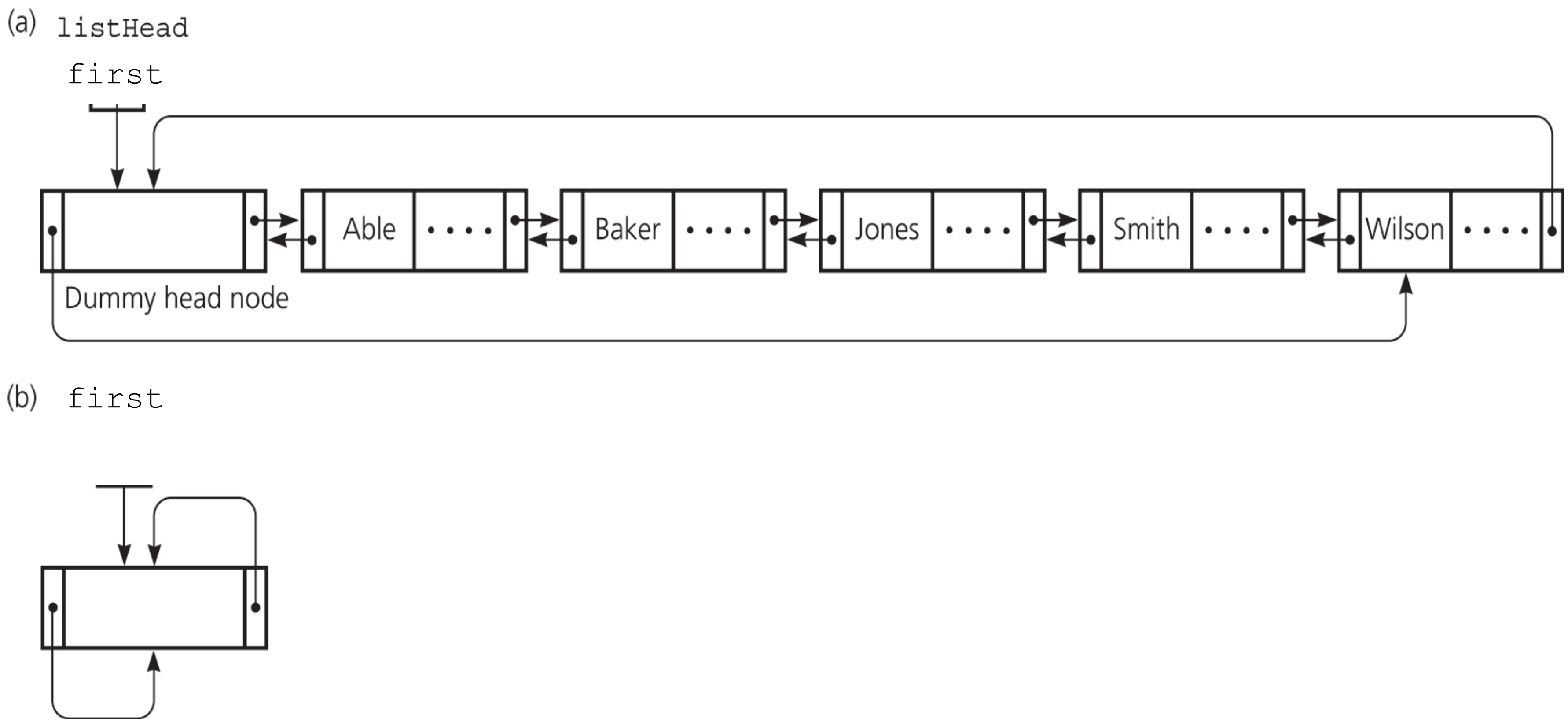


A circular linked list

Circular Doubly Linked Lists

- Circular doubly linked list
  - `prev` pointer of the dummy head node points to the last node
  - `next` reference of the last node points to the dummy head node
  - No special cases for insertions and deletions

Circular Doubly Linked Lists



(a) A circular doubly linked list with a dummy head node

(b) An empty list with a dummy head node

Processing Linked Lists Recursively

- Recursive strategy to display a list
  - Write the first node of the list
  - Write the list minus its first node
- Recursive strategies to display a list backward
  - `writeListBackward` strategy
    - Write the last node of the list
    - Write the list minus its last node backward