

Programming and Algorithms

COMP1038.PGA

Session 15:

Singly Linked List

Overview

- Linked list
 - Introduction
 - Creation
 - Insertion
 - Deletion
 - Printing
 - Searching
 - Application



Introduction

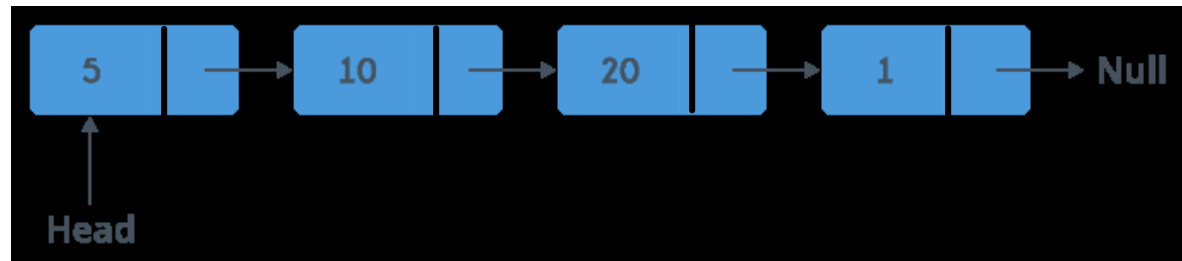
- Lists are linear data structures which store elements of the list one after another.
- Unlike arrays, we can add/remove elements without having to re-create the entire data structure.
- Insert/remove elements from anywhere in the list.
- Access elements anywhere in the list, but slower than arrays.



Introduction cont...

- Linked list is a list of elements that are connected to each other by pointers.

```
struct Node
{
    int data;
    struct Node *next;
}
```



Comparison with Array

- Collection of items stored at continuous memory location.

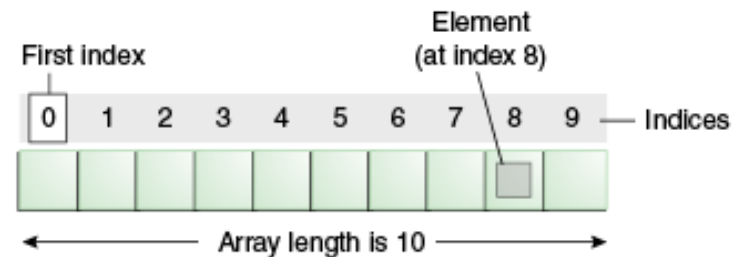
```
#include <stdio.h>

int main()
{
    int arr[3] = {1, 2, 3};

    int i = 0;
    for(i = 0; i < 3; i++)
    {
        printf("#%d: %d\n", i, arr[i]);
    }

    // random access
    printf("\n\nRandom access #%d: %d\n", 2, arr[2]);

    return 0;
}
```



Source: <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html>

```
[z2017233@CSLinux Desktop]$ gcc c_test.c -o c_test
[z2017233@CSLinux Desktop]$ ./c_test
#0: 1
#1: 2
#2: 3

Random access #2: 3
```

Comparison with Array

cont...

- Linked list:
 - Can be of any size as long as memory permits
 - Insertion and deletion of data is easier
- Array:
 - Size is fixed i.e. if array is created, cannot change size again during execution of program.
 - Insertion and deletion of data is difficult



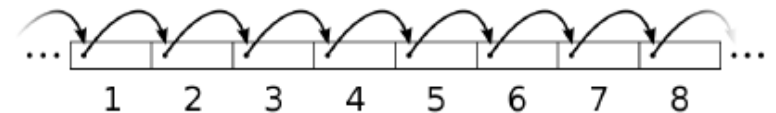
Comparison with Array

cont...

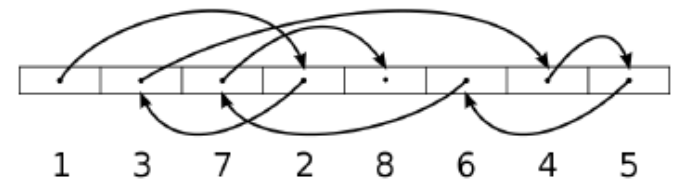
- Linked list: sequential access

```
// This function prints contents of linked list starting from head
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}
```

Sequential access



Random access



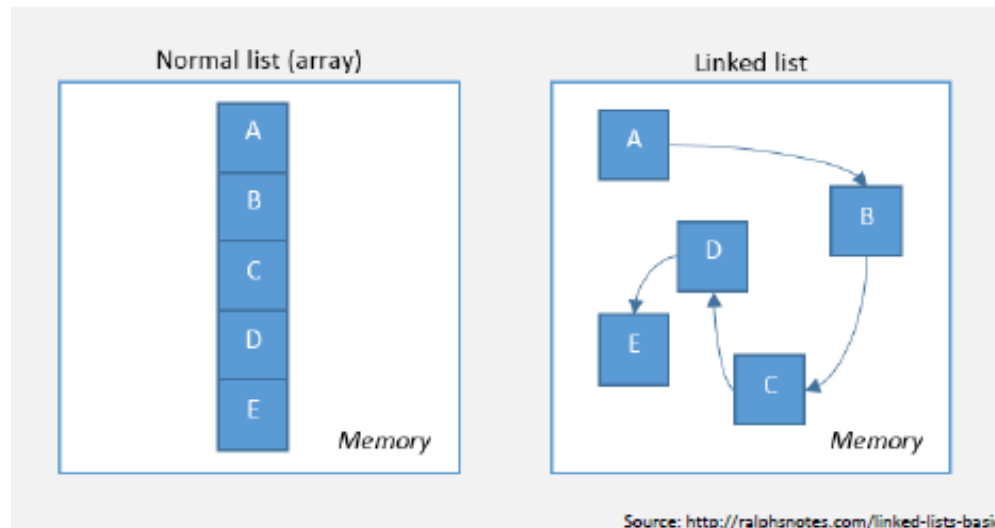
Source: https://en.wikipedia.org/wiki/Random_access

- Array: random access

```
// random access
printf("\n\nRandom access #d: %d\n", 2, arr[2]);
```

Comparison with Array cont...

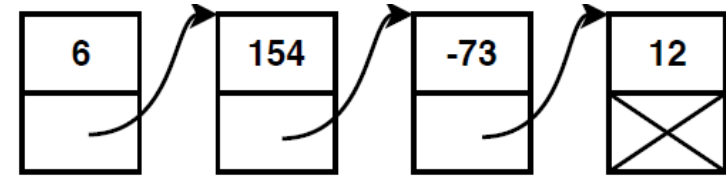
- Memory allocation:



Types

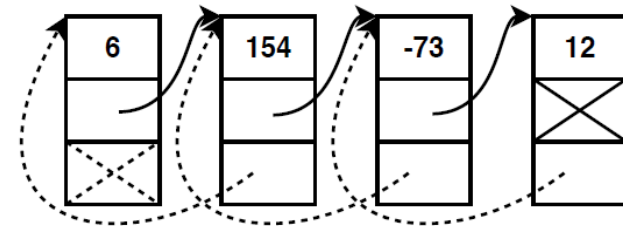
■ Singly Linked list

- A pointer to the next element of the list.
- Indicate end of list with NULL in the last pointer.
- Can only navigate the list in one direction.
- Accessing previous element requires traversing from the start of the list again.



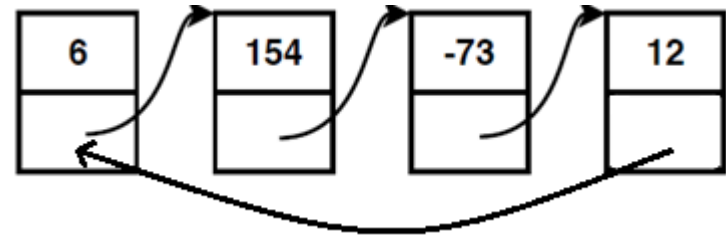
■ Doubly Linked list

- Pointer to next element as with singly-linked list.
- Pointer to previous element as well.
- Can access previous element just by using previous pointer.
- More efficient navigation but more complex algorithms and larger storage requirements



■ Circular Linked list

- A pointer to the next element of the list.
- End node of the list points to the first node of the list
- Can only navigate the list in one direction.



Creation

```
struct Node
{
    int data;
    struct Node *next;
}
```

```
int main()
{
    struct Node *list =
        malloc(sizeof(struct Node));
    ...
}
```



Insertion

■ Insertion at the beginning

/* Given a reference (pointer to pointer) to the head of a list
and an int, inserts a new node on the front of the list. */

```
void insertBeginning(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct
Node));
```

/* 2. put in the data */

```
new_node->data = new_data;
```

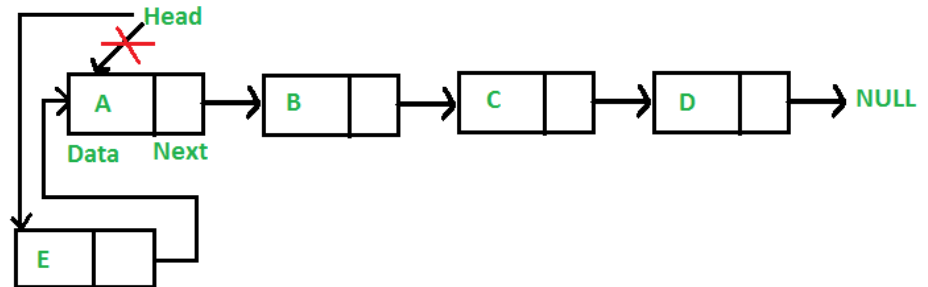
/* 3. Make next of new node as head */

```
new_node->next = (*head_ref);
```

/* 4. move the head to point to the new node */

```
(*head_ref) = new_node;
```

```
}
```



Insertion cont...

■ Add a node after a given node:

/* Given a node prev_node, insert a new node after the given prev_node */

void insertAfter(struct Node* prev_node, int new_data)

{

/*1. check if the given prev_node is NULL */

if (prev_node == NULL)

{

printf("the given previous node cannot be NULL");

return;

}

/* 2. allocate new node */

struct Node* new_node =(struct Node*) malloc(sizeof(struct Node));

/* 3. put in the data */

new_node->data = new_data;

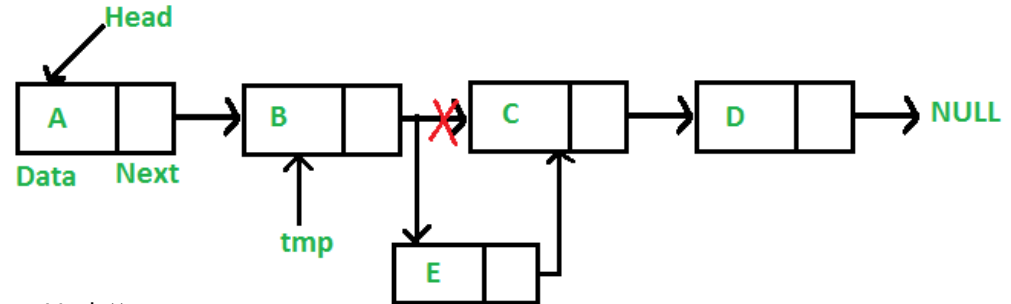
/* 4. Make next of new node as next of prev_node */

new_node->next = prev_node->next;

/* 5. move the next of prev_node as new_node */

prev_node->next = new_node;

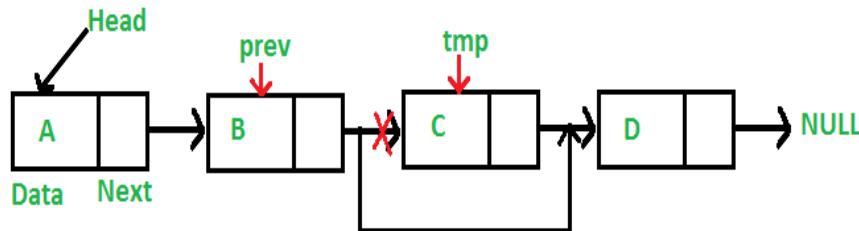
}



Deletion

■ Delete a node with given key

- 1) Find previous node of the node to be deleted.
- 2) Change the next of previous node.
- 3) Free memory for the node to be deleted.



```
/* Given a reference (pointer to pointer) to the head of a list
and a key, deletes the first occurrence of key in linked list */
```

```
void deleteNode(struct Node **head_ref, int key)
```

```
{
```

```
    // Store head node
```

```
    struct Node* temp = *head_ref, *prev;
```

```
    // If head node itself holds the key to be deleted
```

```
    if (temp != NULL && temp->data == key)
```

```
    {
```

```
        *head_ref = temp->next; // Changed head
```

```
        free(temp);           // free old head
```

```
        return;
```

```
    }
```

```
    // Search for the key to be deleted, keep track of the
```

```
    // previous node as we need to change 'prev->next'
```

```
    while (temp != NULL && temp->data != key)
```

```
    {
```

```
        prev = temp;
```

```
        temp = temp->next;
```

```
    }
```

```
    // If key was not present in linked list
```

```
    if (temp == NULL) return;
```

```
    // Unlink the node from linked list
```

```
    prev->next = temp->next;
```

```
    free(temp); // Free memory
```

```
}
```

Deletion cont...

■ Delete a Linked List node at a given position

/ Given a reference (pointer to pointer) to the head of a list and a position, deletes the node at the given position */*

```
void deleteNode(struct Node **head_ref, int position)
```

```
{
```

```
    // If linked list is empty
```

```
    if (*head_ref == NULL)
```

```
        return;
```

```
    // Store head node
```

```
    struct Node* temp = *head_ref;
```

```
    // If head needs to be removed
```

```
    if (position == 0)
```

```
{
```

```
    *head_ref = temp->next; // Change head
```

```
    free(temp); // free old head
```

```
    return;
```

```
}
```

```
    // Find previous node of the node to be deleted
```

```
    for (int i=0; temp!=NULL && i<position-1; i++)
```

```
        temp = temp->next;
```

```
    // If position is more than number of nodes
```

```
    if (temp == NULL || temp->next == NULL)
```

```
        return;
```

```
    // Node temp->next is the node to be deleted
```

```
    // Store pointer to the next of node to be deleted
```

```
    struct Node *next = temp->next->next;
```

```
    // Unlink the node from linked list
```

```
    free(temp->next); // Free memory
```

```
    temp->next = next; // Unlink the deleted node from list
```

```
}
```

Printing

// This function prints contents of linked list starting from
// the given node

```
void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}
```



Searching an item

- 1) Initialize a node pointer, current = head.
- 2) Do following while current is not NULL
 - a) current->key is equal to the key being searched return true.
 - b) current = current->next
- 3) Return false

```
/* Checks whether the value x is present in linked list
*/
bool search(struct Node** head_ref, int key)
{
    struct Node* temp = *head_ref; // Initialize current
    while (temp != NULL)
    {
        if (temp ->key == key)
            return true;
        temp = temp ->next;
    }
    return false;
}
```



Linked list: application

- **Applications of linked list in computer science:**
 - Implementation of stacks and queues
 - Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
 - Dynamic memory allocation : We use linked list of free blocks.
 - Maintaining directory of names
 - Performing arithmetic operations on long integers
 - Manipulation of polynomials by storing constants in the node of linked list
 - representing sparse matrices
- **Applications of linked list in real world:**
 - Image viewer – Previous and next images are linked, hence can be accessed by next and previous button.
 - Previous and next page in web browser – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
 - Music Player – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

The End

