ESC101N
Fundamentals of Computing

Arnab Bhattacharya
arnabb@iitk.ac.in

Indian Institute of Technology, Kanpur
http://www.iitk.ac.in/esc101/

1st semester, 2010-11

Tue, Wed, Fri 0800-0900 at L7

Linked Lists

- *Linked lists* are *dynamic data structures*
- Data structure: They simulate lists of elements
- Dynamic: Size of a linked list grows or shrinks during the execution of a program and is just right
- Advantage: It provides flexibility in inserting and deleting elements by just re-arranging the links
- Disadvantage: Accessing a particular element is not easy

Linked Lists

- *Linked lists* are *dynamic data structures*
- Data structure: They simulate lists of elements
- Dynamic: Size of a linked list grows or shrinks during the execution of a program and is just right
- Advantage: It provides flexibility in inserting and deleting elements by just re-arranging the links
- Disadvantage: Accessing a particular element is not easy
- There are three major operations on linked lists
  1. Insertion
  2. Deletion
  3. Searching

Structure of a linked list

- A linked list contains data
- Each element of the list *must* also *link* with the next element
- Therefore, a structure containing data and link is created
  - Data can be anything
- The link or pointer is to the same type of structure again

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

- This is called a self-referential pointer

Structure of a linked list

- A linked list contains data
- Each element of the list *must* also *link* with the next element
- Therefore, a structure containing data and link is created
  - Data can be anything
- The link or pointer is to the same type of structure again

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

- This is called a self-referential pointer
- A linked list is simply a chain of such nodes
- The beginning of the list is maintained as a pointer to node (generally called `head`)

Structure of a linked list

- A linked list contains data
- Each element of the list *must* also *link* with the next element
- Therefore, a structure containing data and link is created
    - Data can be anything
- The link or pointer is to the same type of structure again

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

- This is called a self-referential pointer
- A linked list is simply a chain of such nodes
- The beginning of the list is maintained as a pointer to node (generally called `head`)
- When a new node (say q) is created (using `malloc`),
    - `q->data` is the desired value
    - `q->next` is `NULL`

Insertion at the beginning

- Create a new node (say q)
- Make `q->next` point to `h`
- Make `head` equal to `q`

Insertion at the beginning

- Create a new node (say q)
- Make `q->next` point to `h`
- Make `head` equal to `q`
- If list is empty, i.e., `head` is `NULL`
    - Make `head` equal to `q`

Insertion at the end

- Create a new node (say `q`)
- Find the last element (say `p`)
- Make `p->next` point to `q`

Insertion at the end

- Create a new node (say `q`)
- Find the last element (say `p`)
- Make `p->next` point to `q`
- If list is empty, i.e., `head` is `NULL`
  - Make `head` equal to `q`

Deletion from the beginning

- Make `p` equal to `head`
- Make `head` equal to `head->next`
- Delete `p` (by using `free`)

Deletion from the beginning

- Make `p` equal to `head`
- Make `head` equal to `head->next`
- Delete `p` (by using `free`)
- If list is empty, i.e., `head` is `NULL`
    - Nothing to do

Deletion from the beginning

- Make `p` equal to `head`
- Make `head` equal to `head->next`
- Delete `p` (by using `free`)
- If list is empty, i.e., `head` is `NULL`
  - Nothing to do
- If list contains only one element
  - Delete `head`
  - `head` is now `NULL`

Deletion from the end

- Find the last element (say `p`)
- While finding `p`, maintain `q` that points to `p`
  - `q` is the node just before `p`, i.e., `q->next` is `p`
- Make `q->next` `NULL`
- Delete `p` (by using `free`)

Deletion from the end

- Find the last element (say `p`)
- While finding `p`, maintain `q` that points to `p`
  - `q` is the node just before `p`, i.e., `q->next` is `p`
- Make `q->next` `NULL`
- Delete `p` (by using `free`)
- If list is empty, i.e., `head` is `NULL`
  - Nothing to do

Deletion from the end

- Find the last element (say `p`)
- While finding `p`, maintain `q` that points to `p`
  - `q` is the node just before `p`, i.e., `q->next` is `p`
- Make `q->next` `NULL`
- Delete `p` (by using `free`)
- If list is empty, i.e., `head` is `NULL`
  - Nothing to do
- If list contains only one element
  - Delete `head`
  - `head` is now `NULL`

Searching a node (insert after, delete after)

- Make `p` equal to `head`
- While `p->data` not equal to the data that is being searched, make `p` equal to `p->next`

Searching a node (insert after, delete after)

- Make `p` equal to `head`
- While `p->data` not equal to the data that is being searched, make `p` equal to `p->next`
- Using search, insert after and delete after operations can be implemented

Searching a node (insert after, delete after)

- Make `p` equal to `head`
- While `p->data` not equal to the data that is being searched, make `p` equal to `p->next`
- Using search, insert after and delete after operations can be implemented
- Insert after `p`
  - Create a new node q
  - Make `q->next` equal to `p->next`
  - Make `p->next` equal to q

Searching a node (insert after, delete after)

- Make p equal to head
- While p->data not equal to the data that is being searched, make p equal to p->next
- Using search, insert after and delete after operations can be implemented
- Insert after p
  - Create a new node q
  - Make q->next equal to p->next
  - Make p->next equal to q
- Delete after p
  - Call the next node, i.e., p->next as q
  - Make p->next equal to q->next
  - Delete q

Linked list operations I

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Node
{
    int data;   // data of a node: list is made of these elements
    struct Node *next;  // link to the next node
} node;

node *create_node(int val)
{
    node *n;
    n = malloc(sizeof(node));
    n->data = val;
    n->next = NULL;
    return n;
}

void print_list(node *h)
{
    node *p;
    p = h;
    while (p != NULL)
    {
        printf("%d --> ", p->data);
        p = p->next;
    }
    printf("NULL\n");
}
```

Linked list operations II

```
int main()
{
    node *head = NULL;  // head maintains the entry to the list
    node *p = NULL, *q = NULL;
    int v = -1, a;

    printf("Inserting at end\n");

    scanf("%d", &v);
    while (v != -1)
    {
        q = create_node(v);
        if (head == NULL)
            head = q;
        else
        {
            p = head;
            while (p->next != NULL)
                p = p->next;

            p->next = q;
        }
        scanf("%d", &v);
    }

    print_list(head);

    printf("Inserting at beginning\n");
```

Linked list operations III

```
scanf("%d", &v);
while (v != -1)
{
    q = create_node(v);
    q->next = head;
    head = q;
    scanf("%d", &v);
}

print_list(head);

printf("Inserting after\n");

scanf("%d", &v);
while (v != -1)
{
    q = create_node(v);

    scanf("%d", &a);
    p = head;
    while ((p != NULL) && (p->data != a))
        p = p->next;

    if (p != NULL)
    {
        q->next = p->next;
        p->next = q;
    }
```

Linked list operations IV

```
        scanf("%d", &v);
    }

    print_list(head);

    printf("Deleting from end\n");

    if (head != NULL)
    {
        p = head;
        while (p->next != NULL)
        {
            q = p;
            p = p->next;
        }

        q->next = NULL;
        free(p);
    }

    print_list(head);

    printf("Deleting from beginning\n");

    if (head != NULL)
    {
        p = head;
        head = head->next;
        free(p);
```

Linked list operations V

```
    }

    print_list(head);

    printf("Deleting after\n");

    scanf("%d", &a);
    p = head;
    while ((p != NULL) && (p->data != a))
        p = p->next;

    if (p != NULL)
    {
        q = p->next;
        if (q != NULL)
        {
            p->next = q->next;
            free(q);
        }
    }

    print_list(head);

}
```

Uses of linked lists

- Linked lists are used to simulate two very important data structures
  1. Stack
  2. Queue

Uses of linked lists

- Linked lists are used to simulate two very important data structures
  1. Stack
  2. Queue
- Stack: Last-in first-out
- Example: stack of dishes
- Operations
  - Push: insert at the beginning
  - Pop: delete from the beginning

Uses of linked lists

- Linked lists are used to simulate two very important data structures
  1. Stack
  2. Queue
- Stack: Last-in first-out
- Example: stack of dishes
- Operations
  - Push: insert at the beginning
  - Pop: delete from the beginning
- Queue: First-in first-out
- Example: queue of people
- Operations
  - Enqueue: insert at the end
  - Dequeue: delete from the beginning

Variants of linked lists

- The simple version is called a singly linked list

Variants of linked lists

- The simple version is called a singly linked list
- When a node contains pointers to both previous and next nodes in the list, it is called a doubly linked list

Variants of linked lists

- The simple version is called a singly linked list
- When a node contains pointers to both previous and next nodes in the list, it is called a doubly linked list
- When the `next` pointer of the last element in the list points back to the `head`, it is called a circular linked list

Variants of linked lists

- The simple version is called a singly linked list
- When a node contains pointers to both previous and next nodes in the list, it is called a doubly linked list
- When the `next` pointer of the last element in the list points back to the `head`, it is called a circular linked list
- When the `previous` pointer of head points to the last element (generally called `tail`), it is called a circular doubly linked list

Variants of linked lists

- The simple version is called a singly linked list
- When a node contains pointers to both previous and next nodes in the list, it is called a doubly linked list
- When the `next` pointer of the last element in the list points back to the `head`, it is called a circular linked list
- When the `previous` pointer of head points to the last element (generally called `tail`), it is called a circular doubly linked list
- Instead of a list, nodes can be arranged in a hierarchical manner also
- It is then called a tree
- In a binary tree, a node points to two *children* nodes
- It may also point to a *parent* node
- The pointer to the structure is maintained as the *root* of the tree

Binary search tree

- A binary search tree is a binary tree
- Every node has two child pointers: `left` and `right`
- The values of all `data` at the tree rooted at the `left` child *must* be *less than or equal* to the value of the `data` at a node
- The values of all `data` at tree rooted at the `right` child *must* be *greater than* the value of the `data` at a node

Binary search tree

- A binary search tree is a binary tree
- Every node has two child pointers: `left` and `right`
- The values of all `data` at the tree rooted at the `left` child *must* be *less than or equal* to the value of the `data` at a node
- The values of all `data` at tree rooted at the `right` child *must* be *greater than* the value of the `data` at a node
- It facilitates faster searching of a value
  - 9 can be searched in 3 steps