Linked Lists

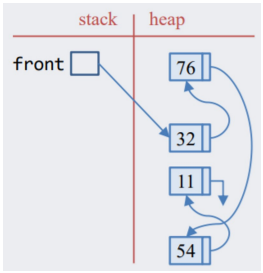
October 4, 2019 8:08 AM

Unordered arrays

- time complexity of inserting an item into an array without holes, when order does not matter

Linked list nodes

- linked list is a dynamic data structure that consists of nodes linked together
- node is a data structure that contains data, and the location (address of the next node)
- data portion of a node can contain one or more items or structures
- Linked list is chain of nodes where eachnode indicates where in (heap)memory the next item can be found
- reached end when next node is null



Modifying linked lists

- **Inserting item**
  - new nodes created using *malloc()*
    - ex. 32 -> 76 -> 54 -> 11
    - want to insert 22 between 74 and 54

      struct node p; //assign p to node 76*
      struct node temp;*
      //temp pointer points to new node
      temp = (struct node)malloc(sizeof(structnode));*
      temp -> data = 22;
      temp -> next = p ->next;
      p -> = temp;
- **Removing item**
  - removed nodes using *free()*
    - ex. 32 -> 76 -> 22 -> 54 -> 11
    - need to free 22 and reassign 76
    - has *p* pointer to 76 and *temp* pointer to 22

      p -> next = temp -> next;
      freet(temp);
      temp = NULL;
- **Advantages**
  - nodes can be aded and removed at runtime
  - size of list does not need to be "guessed" ahead of time
  - to access particular node, may need to traverse list to reach
    - O(n)
  - have aditional overhead to store pointers
  - harder to program, debug and test

Linked list Traversal

- can be done with node pointer var
- ex. to access to 4th element

```
node* ptr = front;
int i;
for (i = 0; i < 4; i++) {
        ptr = ptr->next;
}
//exited loop, ptr references 4th node
printf("%d", ptr->data);
```

- access last item

```
while (ptr != NULL) {
        //do something with ptr --> data
ptr = ptr --> next
}
while (ptr != NILL && ptr -- > data != 15);
```

- Example

```
int main() {
  Player* head_list1 = NULL;
  Player* head_list2 = NULL;
  Player gretzky = {99,
                    "Wayne Gretzky", NULL};
```

```
Player* insertAtHead(Player* head, int player_number, char* player_name) {
  Player new_node;

  new_node.jersey_number = player_number;
  new_node.name = player_name;
```

```
int main() {
  Player* head_list1 = NULL;
  Player* head_list2 = NULL;
  Player gretzky = {99,
                    "Wayne Gretzky", NULL};

  // example 1
  head_list1 = &gretzky;
  displayList(head_list1);

  // example 2
  head_list2 = insertAtHead(head_list2,
                            22, "Daniel Sedin");
  head_list2 = insertAtHead(head_list2,
                            33, "Henrik Sedin");
  displayList(head_list2);
}
```

```
  Player new_node;

  new_node.jersey_number = player_number;
  new_node.name = player_name;
  new_node.next = head; // point to current head of list

  printf("Node was added.\n\n");
  return &new_node; // the new node becomes the new head of list
}
```

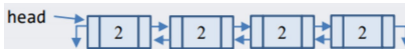
```
typedef struct Player {
  int jersey_number;
  char* name;
  struct Player* next;
} Player;
```

```
void displayList(Player* node) {
  int k = 0;

  while (node) { // loop breaks when node becomes NULL
    printf("Node %d is: %s, Jersey number %d\n",
           k, node->name, node->jersey_number);
    node = node->next;
    k++;
  }

  printf("There are %d node(s) in the list.\n\n", k);
}
```

- *insertAthead* should insert new node at head of a possibly empyt list, return node head
- for loops *for(p = node; p != NULL; p -> next)*
- iterate through linked list, starting from head, printin out each node

Comparison of worst case complexities

Operations	Array (unordered)	Array (ordered)	Linked List (unordered)	Linked list (ordered)
Instert at front	O(1)	O(n)	O(1)	O(1)
Insert at back using head ptr	O(1)	O(1)	O(n) (traversal)	O(n) (traversal)
Insert after current position	O(1)	O(n)	O(1)	O(1)
Search for a value	O(n)	O(n)	O(n)	O(n)
Remove at current position	O(1)	O(n)	O(n)	O(n)

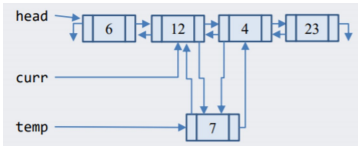


Doubly linked list

- operations at end of list have poor complexity involving complete traversal of list
- use a tail pointer for very little additional overhead
- links to previous node in list, allows traversal or access towards front of the list
- provides access to previous and next nodes from single pointer
- **Insertion**

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

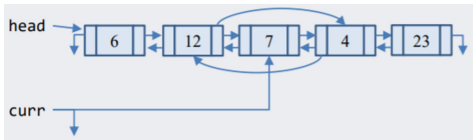
```
Node* curr, * temp;
... // move curr into place
temp = (Node*) malloc(sizeof(Node));
temp->data = 7;
temp->prev = curr;
temp->next = curr->next;
curr->next->prev = temp;
curr->next = temp;
```



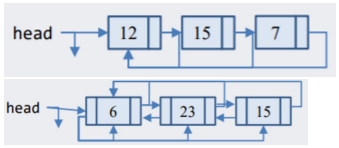

```
typedef struct node {
        int data;
        struct node * next;
} node;
typedef struct LinkedList {
        node* front;
        node* tail;
        int size;
} LinkedList;
void initializeList(LinkedList* ll) {
        ll -> front = NULL;
        ll -> tail = NULL;
        ll -> size  = 0;
}
main {
        node* head;
        node* tail;
        LinkedList myList;
        initializeList(&myList)
}
```

- **Removal**

```
Node* curr;
... // move curr to the node to be removed
curr->next->prev = curr->prev;
curr->prev->next = curr->next;
free(curr);
curr = NULL;
```



- **Circular linked lists**
  - last node in list points to first node
  - reached end in traversal when address of next is address of front
  - can insert in middle of circular singly linked list
    - inserting at the head? need to iterate pointer to last node
  - circular doubly linked list
    - last node points to first no




```
Node* ptr, * newnode;
ptr = head;
while (ptr->next != head)
  ptr = ptr->next;
newnode = (Node*) malloc(sizeof(Node));
newnode->data = 4;
newnode->next = head;
ptr->next = newnode;
head = newnode;
```

- null-terminated
  - singly linked
  - front/head
- circular
  - doubly-linked
  - stront/head + head/tail
  - last node in list points bak to first node, matches address of first one
  - compare head address to address in last pointer to check if at end
- insertion in the middle of a circular singly-linked list
- circular doubly linked list
  - last node in listh points to first noed
  - first node points to last node