UCLA Samueli Computer Science

CS32: Introduction to Computer Science

Discussion Week 4

Junheng Hao, Ramya Satish

Feb 1, 2019

Announcement (Week 4)

UCLA Samueli Computer Science

- Homework 2 is due on next Tuesday, February 5.
  - Make sure you understand the requirements and also read the FAQ.
- About midterm this Wednesday (Part 1)
  - Grades will be released soon by Prof. David Smallberg.

Outline Today

UCLA Samueli Computer Science

- **Linked List**
  - Properties of linked list: Review
  - Doubly linked list
  - Sorted linked list
  - Reverse a linked list
- **Stack and Queues**
  - Implementation and applications
- **Inheritance and Polymorphism (Probably not this time)**
  - Examples and applications

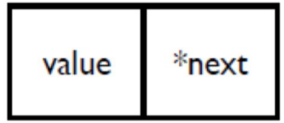
- **Group Exercises**

Linked List: Review

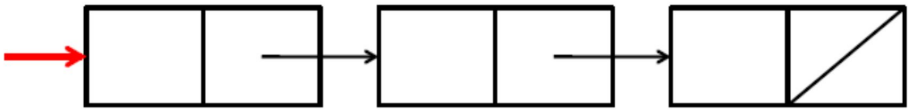
Basis

UCLA Samueli Computer Science

- Minimum Requirement
  - Key component as unit: Node (with `value` and `pointer` to next node)
  - Head pointer → points to the first term
  - Loop-free (except in some special case: circular listed list)
- Regular operations
  - Insertion
  - Search
  - Removal
- Pros and cons
  - Efficient insertion, flexible memory allocation, simple implementation
  - High complexity of search




```
typedef int ItemType;
Struct Node
{
      ItemType value;
      Node *next;
};
```

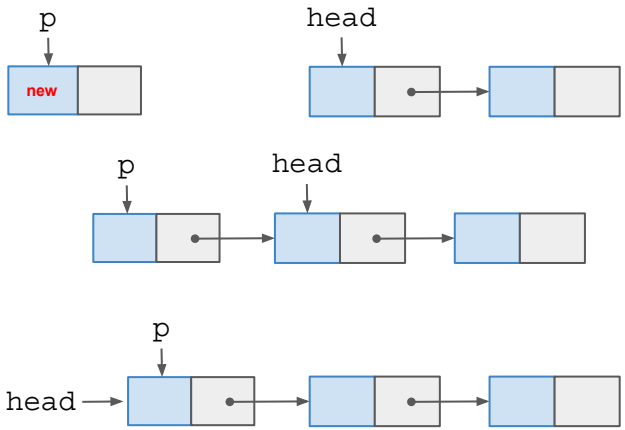


Linked List

Insertion: Add a new node to a list

UCLA Samueli Computer Science

- Example: Insert as head in a list
- Steps
  a) Create a new node and call the pointer `p`
  b) Make its `next` pointer point to the first item
  c) Make the head pointer to the new node



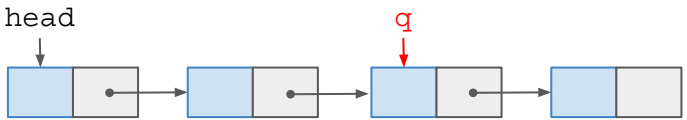

```
//Skeleton: Linked list insertion
//====================================

//insert as head
p->next = head;
Head = p;

//insert after end: End node: q
q->next = p;
p->next = nullptr;

//insert in the middle: node q
p->next = q->next;
q->next = p;
```

Linked List

Search

UCLA Samueli Computer Science

- Steps
  a) Find matched node and return
  b) If no match, return NULL




```
// Skeleton Code: Linked list search
// ==================================


Node* Search(int key, Node* head){
Node *q = head;
while(q != NULL)
{
     if(q -> value != key) q = q -> next;
     else return q;
}
return NULL;
}
```

Linked List

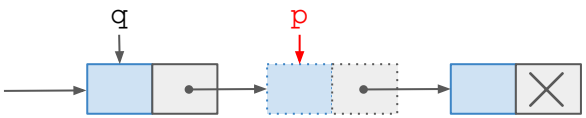
Removal

UCLA Samueli Computer Science

- Remember to set the previous node `q`'s `next` pointer to point the next node of `p`

  `q->next = p->next;`

  `Delete p`
- What if `p == head`? What if `p` prints to the last node in the linked list?




```
// Skeleton Code: Linked list removal
// ===================================
void remove(int valToRemove, Node* head) {
     Node *p = head, *q = NULL;
     while (p != NULL) {
          if (p->value == valToRemove)
               break;
          q = p;
          p = p->next;}
     if (p == NULL) return;
     if (p == head) //special case
          head = p->next;
     else
          q->next = p->next;
     delete p;
}
```

Linked List

Conclusion

UCLA Samueli Computer Science

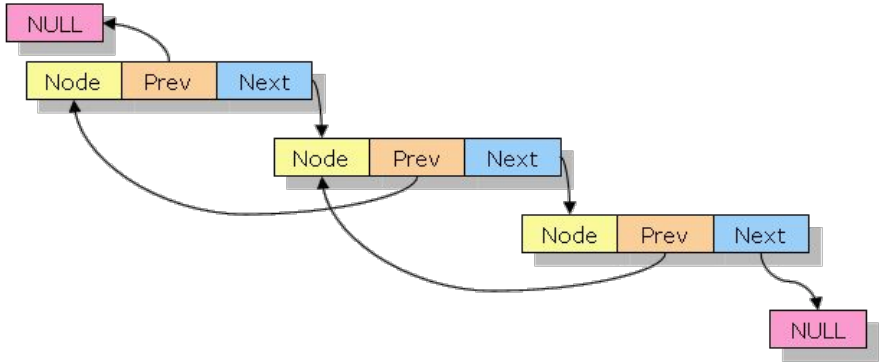
- Pros:
  - Efficient insertion (add new data items)
  - Flexible memory allocation
- Cons:
  - Slow search (search is more important than insertion and removal in real situations)

- Many variations
  - Doubly linked lists
  - Sorted linked lists
  - Circularly linked lists

Double Linked List

Data structures and properties

UCLA Samueli Computer Science



- A linked list where each node has two pointers:
  - Next – pointing to the next node
  - Prev – pointing to the previous node
- Features
  - head, tail pointers
  - `head->prev = NULL; tail->next = NLL;`
  - `head == tail == NULL` when doubly linked list is empty

```
typedef int ItemType;
Struct Node
{
      ItemType value;
      Node *next;
      Node *prev;
};
```

Double Linked List

Insertion: How many cases to consider?

UCLA Samueli Computer Science

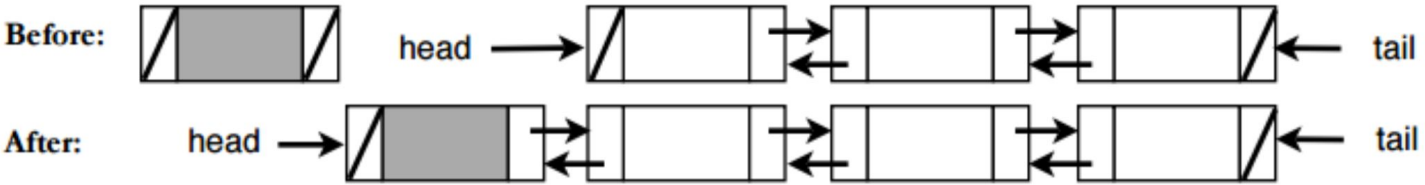
- Four cases:
  - Insert before the head
  - Insert after the tail
  - Insert somewhere in the middle
  - When list is empty ⬅

Double Linked List

Insertion: Before head / After tail

UCLA Samueli Computer Science

- Steps for insertion before head:
  - Set the `prev` of head to the new node `p`
  - Set the `next` of `p` to `head`
  - `p` becomes the new `head`
  - `head->prev = NULL;`
- Steps for insertion after tail:
  - Similar to insertion before head (try it yourself!)



Double Linked List

Insertion: In the middle of the list

UCLA Samueli Computer Science

- Steps for insertion in the middle (after node q):
  - Fix the next node of q first: `Node *r = q->next;`
  - Point both next of q and prev of r to p: `q->next = r->prev = p;`
  - Point both sides of p to q and r respectively: `p->prev = q; p->next = r;`

- You can do that without the help of pointer `r`

```
p->prev = q;
p->next = q->next;
q->next = q->next->prev = p;
```

Double Linked List

Insertion to empty list / Search

UCLA Samueli Computer Science

- Insertion to an empty list

```
head = tail = p;
p->next = p->prev = NULL;
```

- Search in doubly linked list
  - Similar to standard linked list
  - Can be done either from head or tail

Double Linked List

Removal

UCLA Samueli Computer Science

- Removal is more complex!
- Consider the following cases:
  - Check if the node p is the head `(p == head)`. Let this boolean be A.
  - Check if the node is the tail `(p == tail)`. Let this boolean be B.
- Different cases:
  - Case 1 (A, but not B): P is the head of the list and there is more than one node.
  - Case 2 (B, but not A): P is the tail of the list, and there is more than one node.
  - Case 3 (A and B): P is the only node.
  - Case 4 (not A and not B): P is in the middle of the list.

Double Linked List

Removal

UCLA Samueli Computer Science

```
void removeNodeInDLL(Node *p, Node& *head, Node& *tail)
{
     if (p == head && p == tail) //case 3
          head = tail = NULL;
     else if (p == head) {
          //case 1
          head = head -> next;
          head -> prev = NULL; }
     else if (p == tail) {
          //case 2
          tail = tail -> prev;
          tail -> next = NULL; }
     else {
          //case 4
          p -> prev -> next = p -> next;
          p -> next -> prev = p -> prev; }
     delete p;
}
```

Double Linked List

Copy a doubly linked list (and more)

UCLA Samueli Computer Science

- Steps
  - Create head and tail for the new list
  - Iterate through the old list. For each node, copy its value to a new node.
  - Insert the new node to the tail of the new list.
  - Repeat until we have iterated the entire old list.
  - Set `NULL` before head and next of tail.

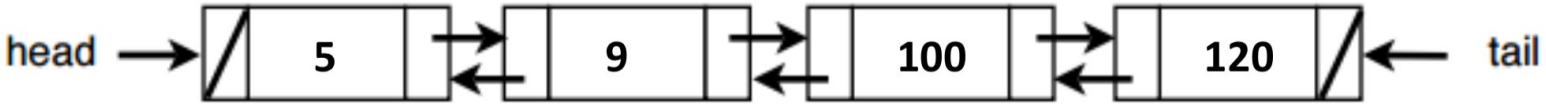
- Tips for linked list problems
  - To draw diagrams of nodes and pointers will be extremely helpful.
  - When copying a linked list, only copy stored values to new nodes. Do not copy pointers.

Sorted Linked List

Motivation and properties

UCLA Samueli Computer Science

- Do we need to search the entire linked list?
- What if we store all values in an ascending sorted (or descending order)?

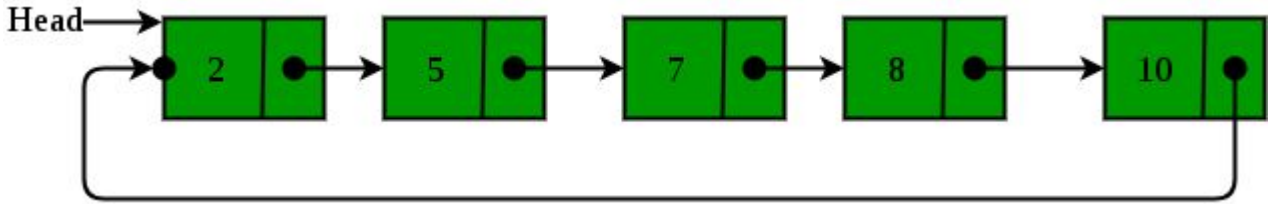


- How do you change insertion function?
  - Find the node `q` whose value is the greatest lower bound to the new node `p`.
- How do you change search function?
  - Early stop when we see a node which stores a value that is larger than key for ascending sorted linked list.
- How do you update removal functions?

Circular Linked List

Motivation and properties

UCLA Samueli Computer Science



- Linked list where all nodes are connected to form a circle.
  - There is no NULL at the end.
  - Can be a singly circular linked list or doubly circular linked list.
- Pros:
  - Any points can be head (starting point).
  - Implementation for queue or Fibonacci Heap.
  - Fit to repeatedly go around the list.
- It is also very tricky though.

Problem: Reverse Linked List

Leetcode questions #206

UCLA Samueli Computer Science

Question: How to reverse a (single) linked list?

Example:

```
Input: 1->2->3->4->5->NULL
Output: 5->4->3->2->1->NULL
```

```
// One possible solution
Node* reverseList(struct ListNode* head)
{
    Node *prev=NULL,*cur=head,*next;
    while(cur) {
        next = cur->next;
        cur->next = prev; prev = cur;
        cur = next;
    }
    return prev;
}
```

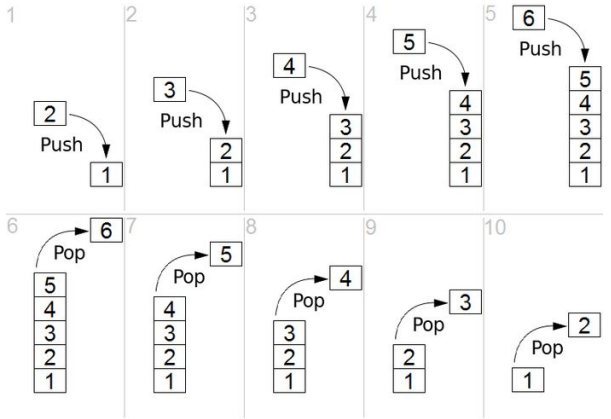
Stack: FILO

Review: Basics

UCLA Samueli Computer Science

- FILO: First In, Last Out
- A standard stack implementation
  - `Push()` and `pop()`
  - Other methods: `top()`, `count()`
- Applications:
  - Stack memory: function call
  - Check expressions: matching brackets
  - Depth-first graph search
- Question: How do you implement stack with linked list / (dynamic) arrays?

```
class Stack
{
public:
      bool push(const ItemType& item);
      ItemType pop();
      bool empty() const;
      int count() const;
private:
      // some features
};
```



Stack

Implement stack with linked list

UCLA Samueli Computer Science

- Container: linked list
- Functions:
  - `push()`: Insert node before head.
  - `pop()`: Remove head and return the head value.
  - `top()`: Read head node.
  - `count()`: Maintain a private `int` member.

- Question: How about using arrays?

Stack

Examples

UCLA Samueli Computer Science

- Given a math expression or text sequence:
  - 6+((5+2)*3-(7+11)*5)*6 → Consider calculation of Reverse Polish Notation
  - Latex: f^{\text{DNN}}(X,\mathbf{W}) = \sigma \left(\mathbf{W} \cdot X + \mathbf{b} \right)

$$f^{\text{DNN}}(X, \mathbf{W}) = \sigma\left(\mathbf{W} \cdot X + \mathbf{b}\right)$$

- How to check the brackets of all types are valid in the sequence? How to calculate expression in Reverse Polish Notation (RPN)?

Regular Expression: 2 + 3 * (5 - 1)

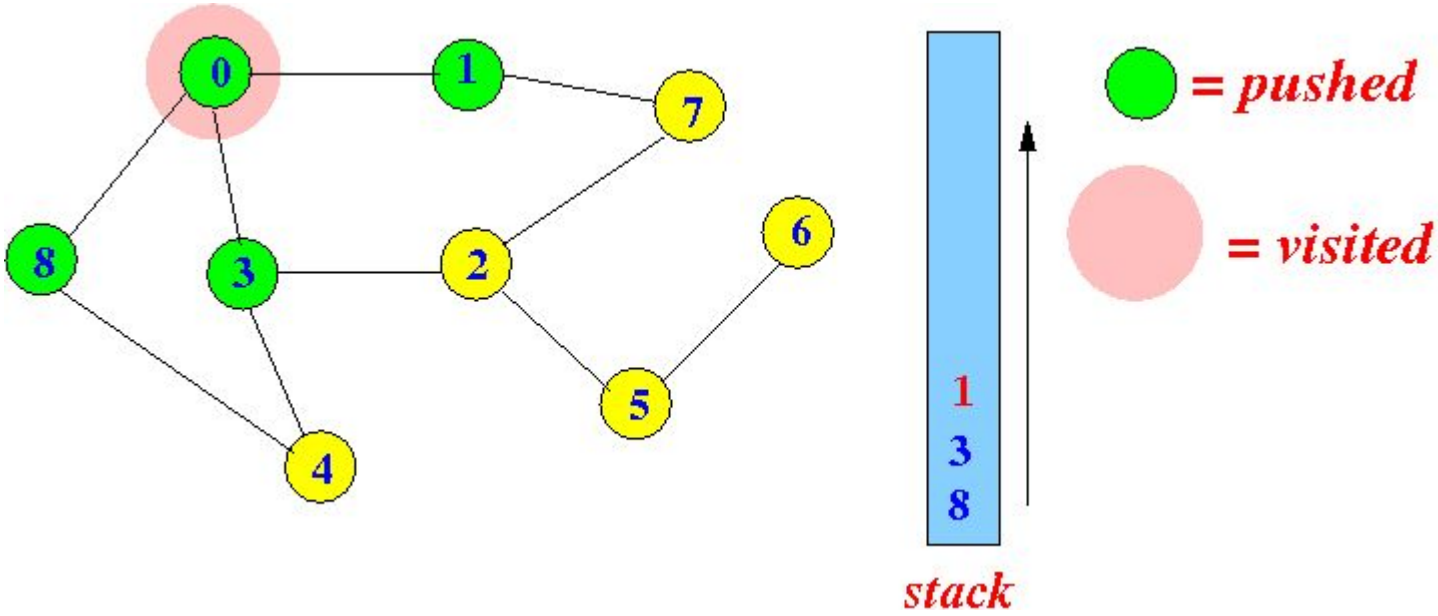
Reverse Polish Notation (RPN): [2] [3] [5] [1] [-] [*] [+]

Stack*

Example: Use stack to implement DFS

UCLA Samueli Computer Science

- Depth-first Search (DFS) on graph (will be later lectures)



Stack*

Example: Use stack to implement DFS

UCLA Samueli Computer Science

```
void Graph::DFS(int s)
{
    vector<bool> visited(V, false);       // Initially mark all vertices as not visited
    stack<int> stack;    // Create a stack for DFS
    stack.push(s);       // Push the current source node
    while (!stack.empty())
    {
        s = stack.top(); // Pop a vertex from stack and print it
        stack.pop();
        // Print the popped item only if it is not visited.
        if (!visited[s])  { cout << s << " "; visited[s] = true;  }
        for (auto i = adj[s].begin(); i != adj[s].end(); ++i)
            if (!visited[*i])
                stack.push(*i);
    }
}
```

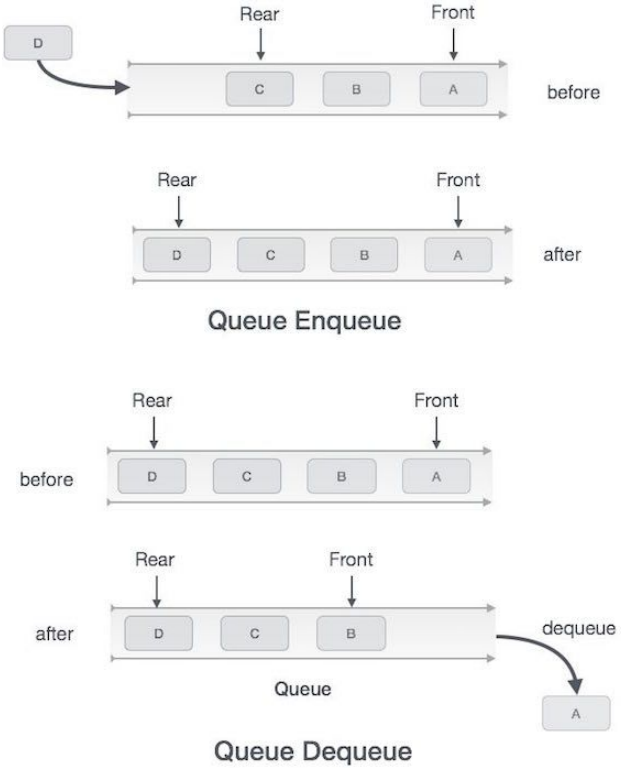
Note: Get all adjacent vertices of the popped vertex s.
If the adjacent has not been visited, then push it to stack.

Queue: FIFO

Review: Basics

UCLA Samueli Computer Science

- FIFO: First In, First Out
- Basic methods:
  - `enqueue(), dequeue()`
  - `front(), back()`
  - `count()`
- Applications
  - Data streams
  - Process scheduling (DMV service request)
  - Breadth-first graph search
- How to implement queue with linked lists or dynamic arrays?



Queue

Extension: Deque(double-ended queue)

UCLA Samueli Computer Science

```
class Deque
{
    public:
        bool push_front(const ItemType& item);
        bool push_back(const ItemType& item);
        bool pop_front(const ItemType& item);
        bool pop_back(const ItemType& item);
        bool empty() const; // true if empty
        int count() const; // number of items
    private:
        int size // Some data structure that keeps the items.
};
```

Question: How to implement `class Deque` with linked lists?

Inheritance & Polymorphism

What's the next?

UCLA Samueli Computer Science

- **Inheritance**
  - Motivation & Definition: Deriving a class from another
  - Construction & Destruction
  - Override a member function

- **Polymorphism**
  - Virtual functions
  - Examples of polymorphism
  - Abstract base class

UCLA Samueli Computer Science

Thank you!

Q & A

Group Exercise

UCLA Samueli Computer Science

Exercise problems from Worksheet 1 & 2 (see “LA worksheet” tab in CS32 website).

Topics today:

- Linked List (Worksheet 1, Question 5-12)

Answers will be posted after discussions.