Data Structure

LEUNG MAN HO (LMH)

Data Structure

- Method of organizing and storing data
  - Organizing objects on desk
  - Schoolbag packing
  - Books in library
  - Supermemory (Memory Palace)
- Enhance data retrieval
- Pillar of algorithms
- Greatly affect complexity
- Big Topic :D

Outline

- Basic ADT
  - ADT: Abstract Data Structure
  - Stack
  - Queue
  - Linked List
- STL
  - Map
  - Vector
  - Set
  - Priority Queue

Stack

- All insertions and deletions are made at one end – top
- Operations:
  - Push: Insert new entry as top
  - Pop: Delete top
  - Top: Query top
- Order of cleaning kitchen dishes
  - Last In First Out (LIFO)

Stack - Push

- Put an element on the top of the stack

```
if (top==MAX_SIZE)      //stack is full
  output "error: stack overflow"
else
  stack[top++] = data;
```

Stack - Pop

- Remove top of the stack

```
if (top==0)     //the stack is empty
  output “Error: stack underflow”
else{
  --top;
}
```

Stack - Top

- Query the element on the top of the stack

```
if (top==0)     //the stack is empty
  output “Stack Empty”
else{
  output stack[top-1];
  return stack[top-1];
}
```

Stack – STL Implementation

- Array-based stack is static allocated
- STL is dynamic allocated – Only scale up with #entries
- `#include<stack>`
- Declaration
  - `stack <type>  Q`
- Basic Operation
  - `Q.push(element);`
  - `Q.top();`
  - `Q.pop();`
  - `Q.empty();`
  - `Q.size();`

Parentheses Balance

- HKOJ01015
- How compilers check your programs for syntax errors
- `[()]` is legal but not `[(])`
- `(())` is legal but not `(()))`

Parentheses Balance

Make an empty stack

- Encounter opening symbol
  - Push into stack
- Encounter closing symbol
  - If stack is empty, error
  - Otherwise, pop the stack
  - If the symbol popped is not the corresponding opening symbol, error.

If stack is not empty at end, error

Expression Evaluation

- Three different but equivalent ways of writing expressions
- Prefix
  - `- + * 2 3 / 5 4 9`
  - operators before their operands.
- Postfix
  - `2 3 * 5 4 / + 9 -`
  - operators written after their operands
- Infix
  - `2 * 3 + 5 / 4 - 9`
  - the usual custom of writing binary operators between their operands
  - Parenthesis may be needed

Expression Evaluation – Prefix & Postfix

- Prefix, postfix evaluation is easy
- Postfix – Read from left to right
  - Operands – Push
  - Operators op
    - Pop two operands out (OP1, OP2)
    - Push (OP1 op OP2) in
- Prefix – Read from right to left
  - Similar to postfix

Expression Evaluation – Postfix Example

- Step 1
  - Sequence: `2 3 * 5 4 / + 9 -`
  - Stack: Empty
- Step 2
  - Sequence: `2 3 * 5 4 / + 9 -`
  - Stack: 2
- Step 3
  - Sequence: `2 3 * 5 4 / + 9 -`
  - Stack: `2 3`

Expression Evaluation – Postfix Example

- Step 4
  - Sequence: 2 3 * 5 4 / + 9 –
  - Stack: 6
- Step 5
  - Sequence: 2 3 * 5 4 / + 9 –
  - Stack: 6 5
- Step 6
  - Sequence: 2 3 * 5 4 / + 9 –
  - Stack: 6 5 4

Expression Evaluation – Postfix Example

- Step 7
  - Sequence: `2 3 * 5 4 / + 9 -`
  - Stack: `6 1.25`
- Step 8
  - Sequence: `2 3 * 5 4 / + 9 -`
  - Stack: `7.25`
- Step 9
  - Sequence: `2 3 * 5 4 / + 9 -`
  - Stack: `7.25 9`

Expression Evaluation – Postfix Example

- Final Step
  - Sequence: `2 3 * 5 4 / + 9 -`
  - Stack: `-1.75`

- Neat and Simple

Expression Evaluation – Infix

- More complex
- Create two stacks
  - operatorStack - for operators and parenthesis
  - operandStack
- Create precedence ordering list
  - * / + –

Expression Evaluation – Infix

```
while not end of line
  1. if char is operand or '('
    push into operandStack
  2. else if char is operator
    2.1 while operatorStack's top precedence >= than char
        2.2 pop from operatorStack
        2.3 pop two operands from operandStack
        2.4 push OP1 op OP2 on the operandStack
  3. else if character is ')'
    2.2 - 2.4 till you encounter '('
```

Expression Evaluation – Infix

```
while stack size > 1
    pop 1 operator (op)
    pop 2 operands (OP1 and OP2)
    push OP1 op OP2 on the operandStack

return top from operandStack
```

Expression Evaluation – Infix Example

- Step 1
  - Sequence: `2 * 3 + 5 / 4 - 9`
  - OperatorStack: `Empty`
  - OperandStack: `Empty`
- Step 2
  - Sequence: `2 * 3 + 5 / 4 - 9`
  - OperatorStack: `2`
  - OperandStack: `Empty`
- Step 3
  - Sequence: `2 * 3 + 5 / 4 - 9`
  - OperatorStack: `2`
  - OperandStack: `*`

Expression Evaluation – Infix Example

- Step 4
  - Sequence: `2 * 3 + 5 / 4 - 9`
  - OperatorStack: `2 3`
  - OperandStack: `*`
- Step 5
  - Sequence: `2 * 3 + 5 / 4 - 9`
  - OperatorStack: `6`
  - OperandStack: `+`
- Step 6
  - Sequence: `2 * 3 + 5 / 4 - 9`
  - OperatorStack: `6 5`
  - OperandStack: `+`

Expression Evaluation – Infix Example

- Step 7
  - Sequence: `2 * 3 + 5 / 4 - 9`
  - OperatorStack: `6 5`
  - OperandStack: `+ /`
- Step 8
  - Sequence: `2 * 3 + 5 / 4 - 9`
  - OperatorStack: `6 5 4`
  - OperandStack: `+ /`
- Step 9
  - Sequence: `2 * 3 + 5 / 4 - 9`
  - OperatorStack: `6 1.25`
  - OperandStack: `+`

Expression Evaluation – Infix Example

- Step 10
  - Sequence: `2 * 3 + 5 / 4 - 9`
  - OperatorStack: `6 1.25`
  - OperandStack: `+`
- Step 11
  - Sequence: `2 * 3 + 5 / 4 - 9`
  - OperatorStack: `7.25`
  - OperandStack: `-`
- Step 12
  - Sequence: `2 * 3 + 5 / 4 - 9`
  - OperatorStack: `7.25 9`
  - OperandStack: `-`

Expression Evaluation – Infix Example

- Final Step
  - Sequence: `2 * 3 + 5 / 4 - 9`
  - OperatorStack: `-1.75`
  - OperandStack: `Empty`

Queue

- Insertions take place at one end – rear
- Deletions take place at the other end – front
- Operations
  - Enqueue – insert new entry at rear
  - Dequeue – delete entry at front
  - Front – Query front element
- Like queuing in shops – First come first serve
- First In First Out (FIFO)

Queue – Enqueue

- Add an element at the end of the queue

```
if (e==MAX)
  output “Error”
else
  queue[e++] = data;
```

Queue – Dequeue

- Remove the element at the front of the queue

```
if (f==e)     //empty queue
  output “error”
else
  ++f;
```

Queue – Front

- Query front element

```
return queue[f]
```

Queue

- Not efficient in terms of memory
- Improvement:
  - Using linked list to implement
  - Circular Queue

Queue – Circular Queue

- Enqueue

```
if(e - f >= size)
  output “ERROR”
else
  queue[e++ % size] = data;
```

- Dequeue

```
if(f == e)
  output “ERROR”
else
  f++;
```

- Front

```
return queue[f % size];
```

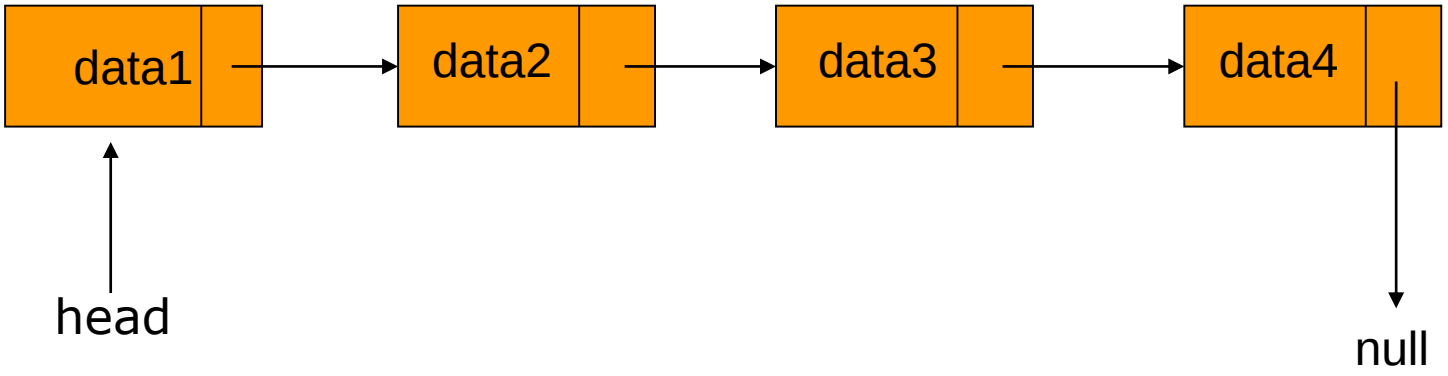
Queue – STL Implementation

- Array-based queue is static allocated
- STL is dynamic allocated – Only scale up with #entries
- `#include<queue>`
- Declaration
  - `queue <type>  Q`
- Basic Operation
  - `Q.push(element);`
  - `Q.front();`
  - `Q.pop();`
  - `Q.empty();`
  - `Q.size();`

Linked List

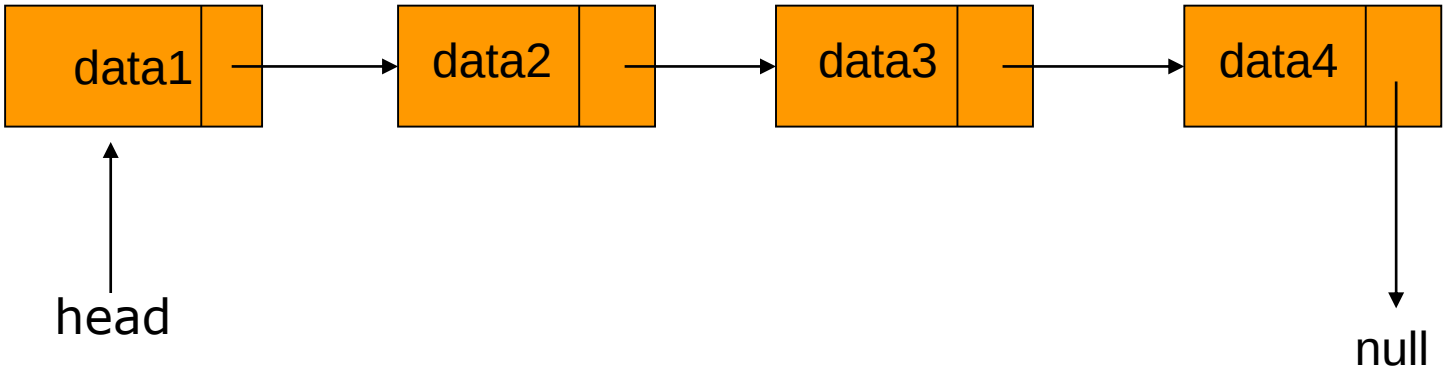
- Data + Pointers
- Operations:
  - Insertion – Placing new entry into particular position
  - Deletion – Removing particular element

Linked List

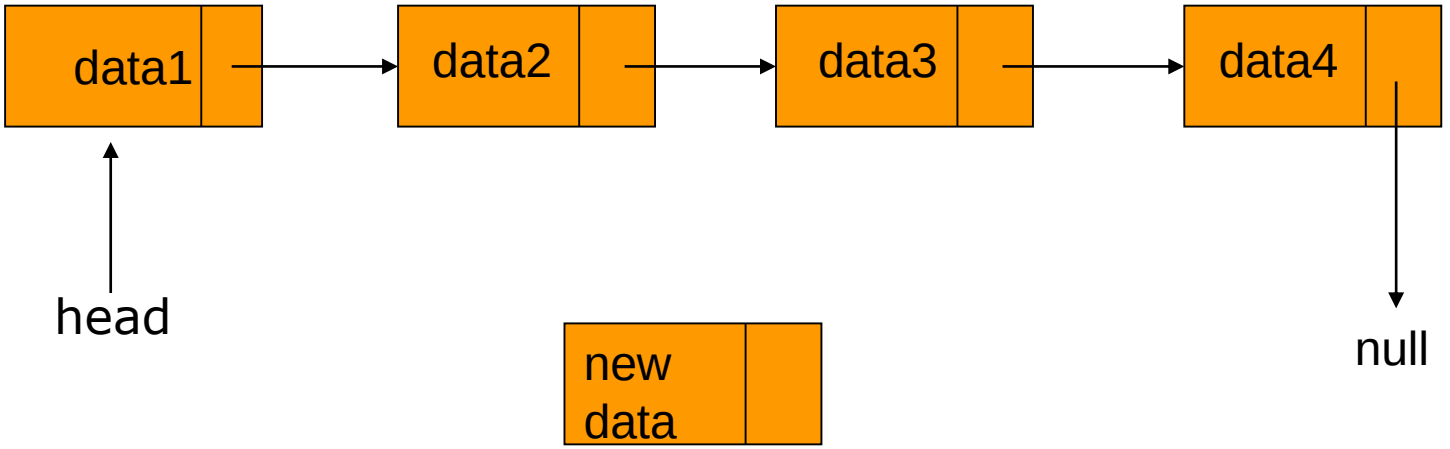


Linked List - Insertion

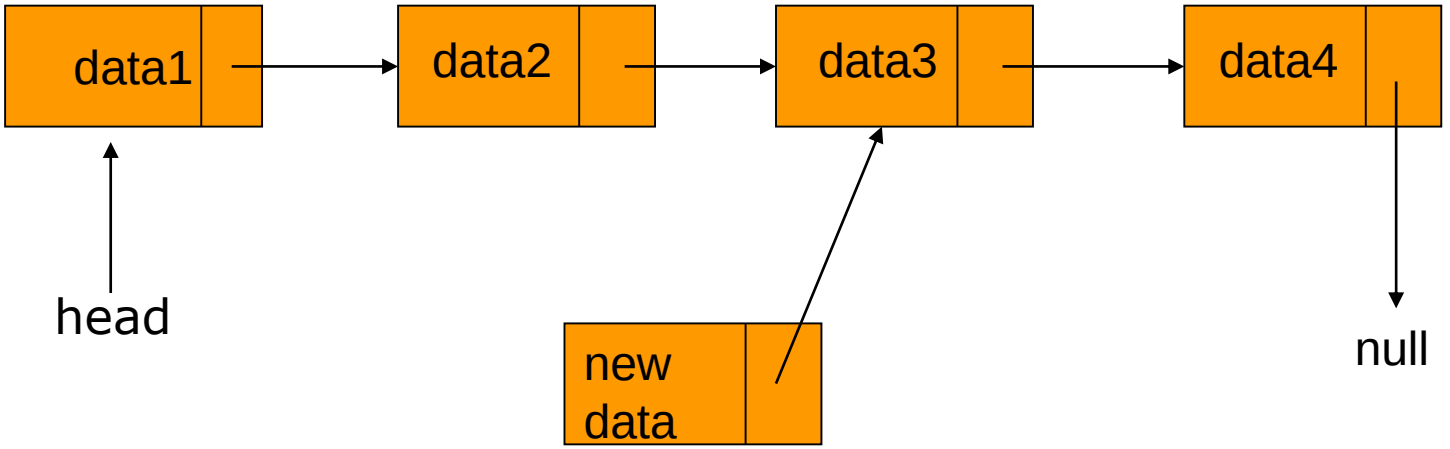
Assume we add a node after node 2



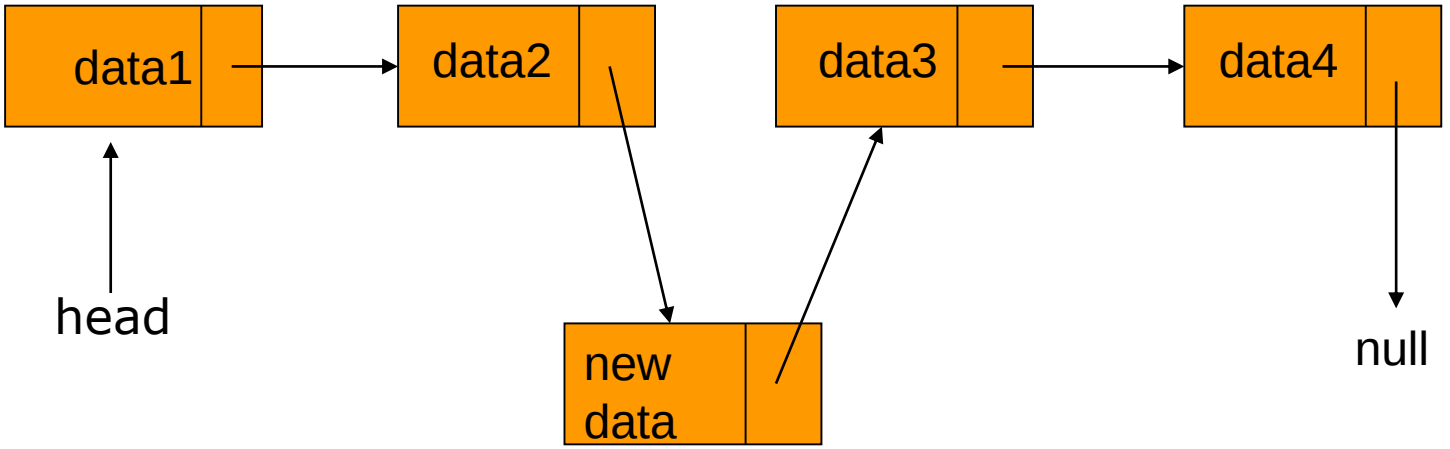
Linked List - Insertion



Linked List - Insertion



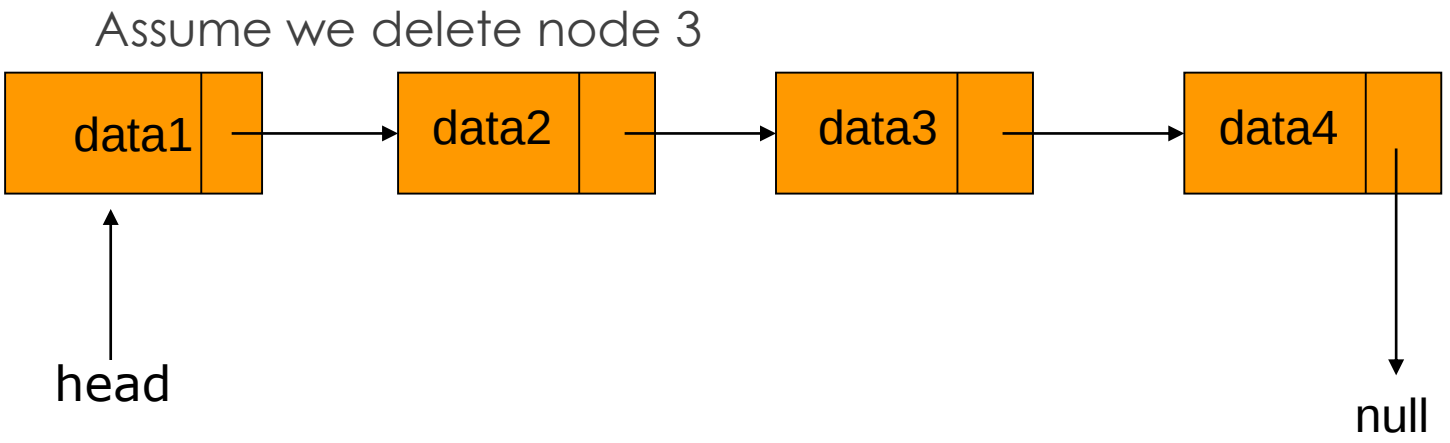
Linked List - Insertion



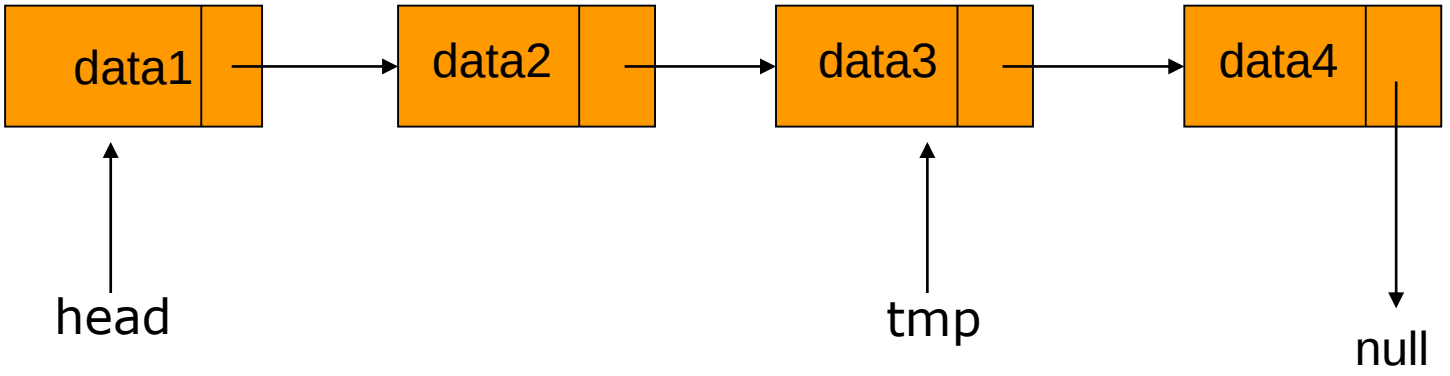
Linked List – Insertion

```
node* tmp = new node;
tmp->data = new_data;
tmp->next = target->next;
target->next = tmp;
```

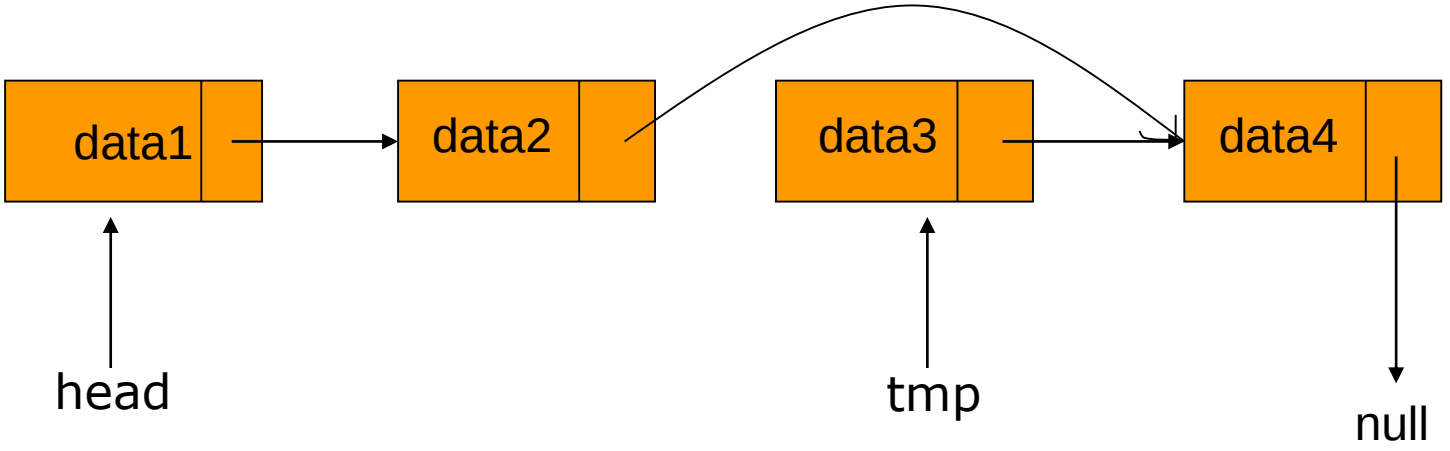
Linked List - Deletion



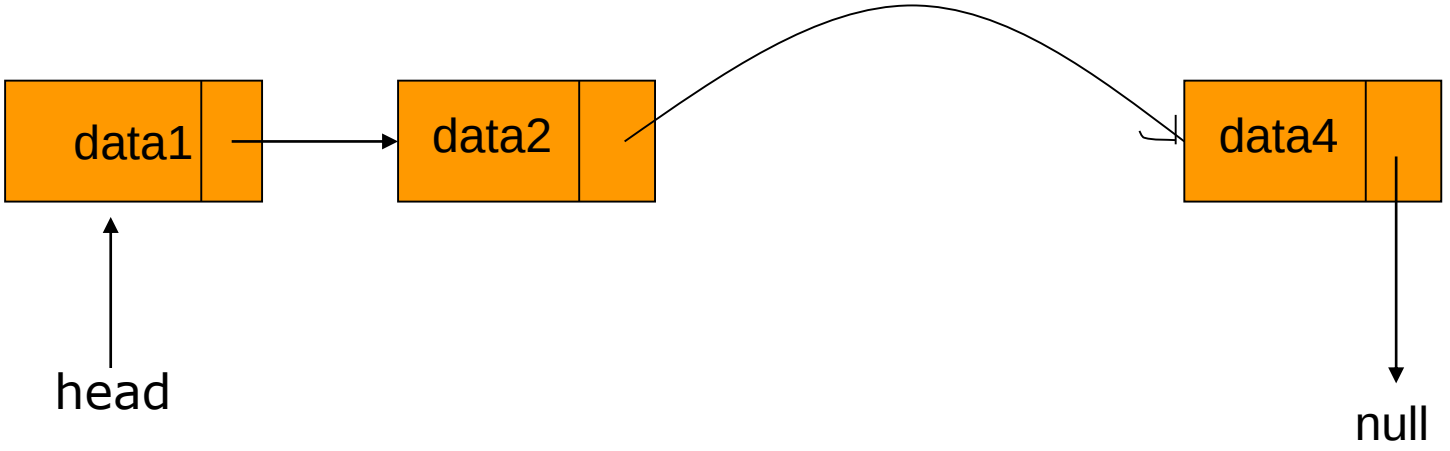
Linked List - Deletion



Linked List - Deletion



Linked List - Deletion



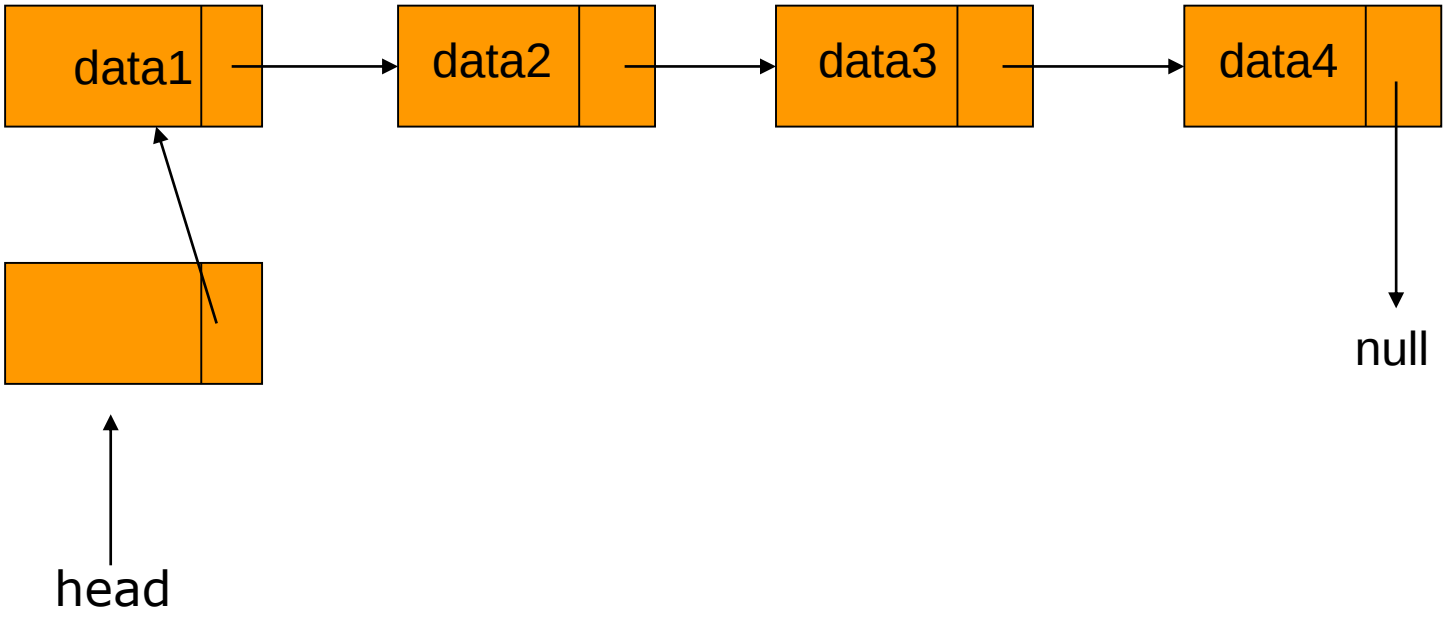
Linked List - Deletion

```
node* tmp = target->next;
target->next = tmp->next;
delete tmp;
```

Linked List

- Troublesome when deleting the first node / inserting before the first node
- Sentinel node
- Doubly Linked List

Linked List – Sentinel Node



Types of Linked List

- Singly-Linked List
- Doubly-Linked List
- Circularly-Linked List

Linked List – Advantage

- Dynamic Allocation
  - When you dunno how large data is
  - Hence could implement dynamic stack, queue
- Fast insertion, deletion compare to array
  - O(1) vs O(N)
- Fast append, delete sequence

Linked List – STL Implementation

- list is doubly linked list in STL (forward_list is singly)
- #include<list>
- Declaration
  - list<int> L
- Basic Operation
  - L.insert();
  - L.delete();
  - L.empty();
  - L.size();
- Traversing
  - for (it=L.begin(); it!=L.end(); ++it) cout << ' ' << *it;

Linked List – STL Implementation

- Example

```
it = mylist.begin();
 ++it;       // it points now to number 2
 mylist.insert (it,10);                          // 1 10 2 3 4 5
 // "it" still points to number 2                      ^
 mylist.insert (it,2,20);                        // 1 10 20 20 2 3 4 5
 --it;       // it points now to the second 20             ^
 std::vector<int> myvector (2,30);
 mylist.insert (it,myvector.begin(),myvector.end());
                                  // 1 10 20 30 30 20 2 3 4 5
```

STL Useful Data Structure

- Map
- Vector
- Set
- Priority Queue

Map – Situation

- Given a wide range of integers, output their freq.
- #Integers <- $[1, 100000]$
- Range <- $[1, 1000000000]$
- Use integer array if you have 4GB memory

Possible Solution – Open hashing

- Refer to Data Structure II (Steven Lau)
- Create N linked list (Say N = prime number 909091)
- Use node to store the freq.
- How to handle new data?
  - Data = 1000000
  - Check for 1000000 in 90909th Linked List
  - Change or Create new entry
- Memory Complexity – O(N)
- Time Complexity for each query – ~O(1)

Alternative Approach – Map

- Too complicated to code? (linked list, pointer, etc…)
  - Actually it's not that complicated with STL vector
- Map in C++ STL
- Easy to use, Fast enough

Map – STL Implementation

- `#include<map>`
- Key - `long long`, Data - `int`

```
map<long long, int> mapping;
mapping[2112434224LL] = 1;
```

- Key – `String(Objective)`, Data – `boolean`

```
map<string, bool> mapping;
mapping[“How are you?”] = true;
```

- Key – `struct`, Data – `struct`, have to define compare

```
map<struct1, struct2> mapping;
mapping[node1] = node2;
```

- Integer mapping – O(log N)
- String mapping – O(L log N)

Map – Basic Operation

- N = number of non-empty element
- `mapping.clear()`
  - O(N)
- Operations often require an iterator(pointer)

```
map<long long,int>::iterator it1, it2;
it1 = mapping.find(int1); it2 = mapping.find(int2); //O(log N)
mapping.erase(it1, it2); //O(|non-empty element between|)
```

- Loop through every non-empty element

```
for (it=mapping.begin(); it!=mapping.end(); ++it)
    cout << it->first << " => " << it->second << '\n';
```

Map – Summary

- Robust
- Perform extra functions than traditional hashing
- Slower than traditional hashing
- Implemented in Red Black Tree
- Memory Complexity – O(N log N)

Vector

- Dynamic Sized Array
- O(1) Add and Delete from the end only (Like Stack)
- Can be randomly accessed

Vector – STL Implementation

- `#include<vector>`
- Example

```
vector<int> v;
v.push_back(1); //O(1)
v.push_back(2);
v[1] = v[0]; //O(1)
v.pop_back();
```

- Memory – O(N)

Vector - Application

- Adjacency List

  `vector<int> head[V]`

  Store all neighboring vertex index

- Hashing

  `vector<type> key[P]`

  Store all information needed for an entry(freq, index, …)

Set

- Weaker version of map (key = value)
- Operation
  - Insert
  - Find
  - Erase
- Compatible with STL Algorithms - set_union, set_intersection, etc

Priority Queue

- STL implemented Max Heap
- Operation
  - Push
    - Insert element inside heap
  - Pop
    - Delete largest element inside heap
- Both operations cost only O(log N)

Priority Queue – Implementation

```
#include<queue>
priority_queue<int> q; // type can be well defined struct
q.push(1); q.push(2);
int1 = q.top(); // int1 = 2
while(!q.empty())
 q.pop();
```

Priority Queue

- A* Searching
- Dijkstra Algorithm (SP Algorithm)
- Prims' Algorithm (Minimum Spanning Tree)
- Various Greedy problems
  - HKOJ01019

Questions?