505 22240 / ESOE 2012 Data Structures: Lecture 4

Linked Lists and Recursion

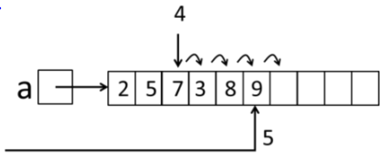
§ Array Lists

- We can store a list of ints as an array.
- Disadvantages:

  ① Insert item at <u>beginning</u> or <u>middle</u>. → time proportional to length of array.

  ② Arrays have a <u>fixed</u> length.

- e.g.




```
class ListExample {
private:
    int* a;
    int lastItem;
public:
    ListExample( ) {
        a = new int[10];
        lastItem = -1;
    }
    void insertItem(int newItem, int location) {
        int i;
        Code A
        for (i = lastItem; i >= location; i--) {
            a[i+1] = a[i];      // move element backward
        }
     -> a[location] = newItem;
        lastItem++;
    }
};
```

- If the array is <u>full</u>, need a new array: Code A

```
int length = sizeof(a) / sizeof(int);
if (lastItem + 1 == length) {
    int* b = new int[2*length];
    for (i = 0; i <= lastItem; i++) {
        b[i] = a[i];        // assign values to b
    }
    delete [ ] a;
    a = b;            // reference a to the new array
}
```

§ Linked Lists (a recursive data type)

- We can avoid these problems by choosing a scheme-like representation of lists.
- A linked list is made up of "nodes".
- Each node has:

  (1) an item.

  (2) a pointer to next node in list.

- e.g.

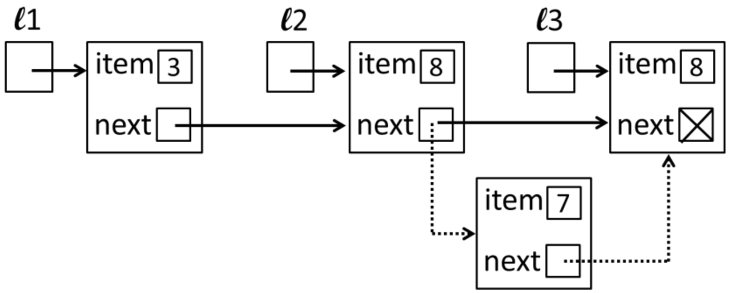
```
class ListNode {
public:
    int item;
    ListNode* next;
    //...
};
```

· Usage

```
ListNode* ℓ1 = new ListNode;
ListNode* ℓ2 = new ListNode;
ListNode* ℓ3 = new ListNode;

ℓ1->item = 3;
ℓ2->item = 8;
ℓ3->item = 8;

ℓ1->next = ℓ2;
ℓ2->next = ℓ3;
ℓ3->next = NULL;
```



◎Node Operations

· Continued within the ListNode class:

```
ListNode(int item, ListNode* next) {
    this->item = item;
    this->next = next;
}   // long constructor
```

```
ListNode(int item) {
    ListNode(item, NULL);
}   // short constructor

ListNode* ℓ1 = new ListNode(3, new ListNode(8, new ListNode(8)));
// no ℓ2, ℓ3
```

◎Advantages over Array Lists

· Inserting item int middle of linked list takes <u>constant</u> time if you have reference to previous node.

· Moreover, list can keep growing until memory runs out.

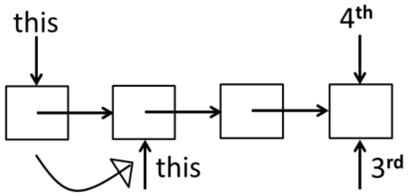
· Inserts a new item after "this" (within the ListNode class):

```
void insertAfter(int item) {
    next = new ListNode(item, next);
}
ℓ2->insertAfter(7);
```

◎Disadvantages

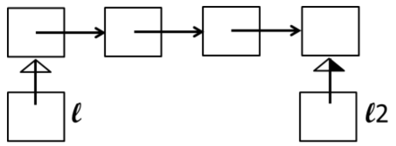
· Finding the n^{th} item of a linked list takes time proportional to n (constant-time on array lists).

· Find a ListNode:



```
ListNode* nth(int position) {
    if (position == 1) {
        return this;
    } else if ((position < 1) || (next == NULL)) {
        return NULL;
    } else {
        return next->nth(position - 1);
    }
}
```

・e.g. `ListNode* ℓ2 = ℓ->nth(4);`



§ Singly Linked Lists

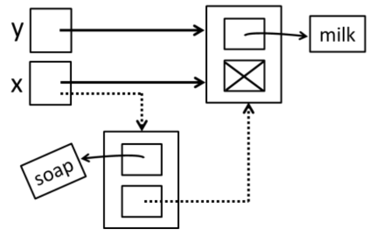
・We can modify the ListNode class:

```
class SListNode {
public:
    string item;
    SListNode* next;
};
```

◎A List Class

・Two problems with the SListNode:

① Suppose x and y are pointers to the same shopping list initially.　We now insert a new item at beginning of the list.




```
x = new SListNode("soap", x);
```

→ y doesn't point to the new item.

→ different SListNodes between x and y.

② How do you represent an empty list?

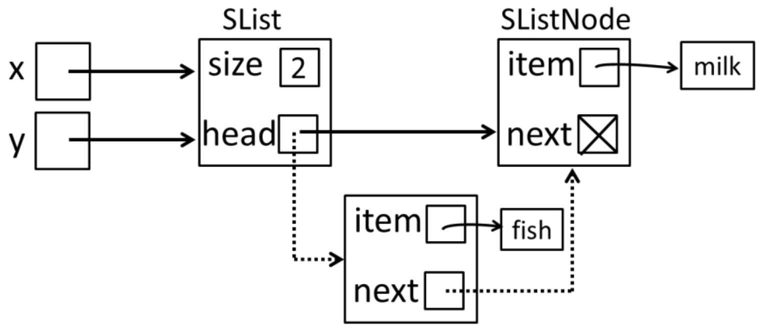
```
x = NULL;
// Run-time ERROR if you call a method on a null object.
```

・e.g.

```
x->nth(1);        // System crash.
```

★Solution: separate SList class and maintains head of list

```
class SList {
private:
    SListNode* head;         // First node in list
    int size;                // Number of items in list
public:
    SList( ) {
        head = NULL;
        // Here's how to represent an empty list.
        size = 0;
    }
    void insertFront(const string& item) {
        head = new SListNode(item, head);
        size++;
    }
};
```



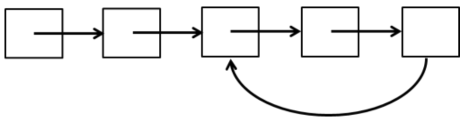
・Now, when an item is inserted at the front of a SList, every pointer to that SList can see the change.

◎The “SList” ADT

・Another advantage of SList class: SList ADT enforces 2 invariants.

① ”size” is always correct.

② list is never circularly linked.



・Both goals accomplished because <u>only</u> SList methods can change the lists.

◎SList ensures this:

① The fields of SList (head and size) are “private”.

② No method of SList returns an SListNode.

◎Inserting/deleting at front of list is easy for SList class

```
void deleteFront( ) {
    if (head != NULL) {
        SListNode* temp = head->next;
        delete head;
        head = temp;
        size--;
    }
}
```

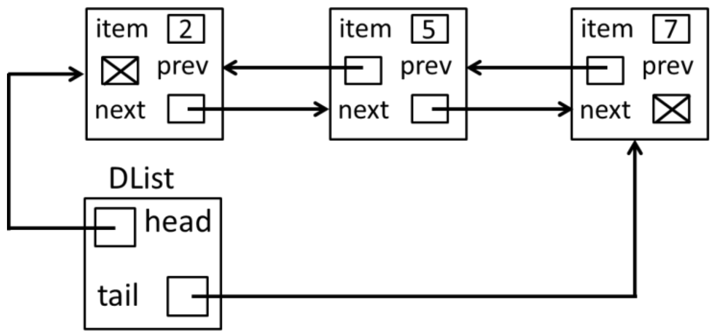
・Inserting/deleting at end of SList takes a long time.

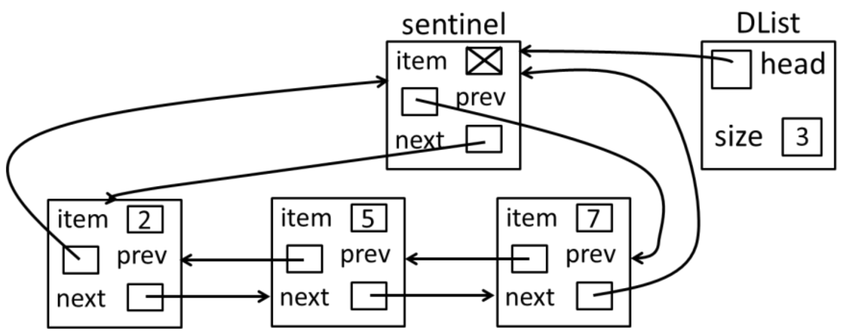
§ Doubly Linked Lists

・A <u>doubly linked list</u> is a list in which each node has a pointer to the previous node, as well as the next node.

```
class DListNode {
public:
    int item;
    DListNode* next;
    DListNode* prev;
};

class DList {
private:
    DListNode* head;
    DListNode* tail;
};
```





· Insert & delete items at both ends in constant running time.

· Removes the tail node (at least 2 items in DList):

```
tail->prev->next = NULL;
DListNode* temp = tail->prev;
delete tail;
tail = temp;
```

◎DList version 2: circularly linked

· Sentinel (dummy) node: a special node that does not represent an item. → should be hidden.

· It represents both the head and the tail of the list.

```
class DList {
private:
    DListNode* head;        // remove tail
    int size;
}
```

◎DList ADT invariants with sentinel:

(1) For any DList* d, d->head ! = NULL. **(always a sentinel)**

(2) For any DListNode* x, x->next ! = NULL.

(3) For any DListNode* x, x->prev ! = NULL.

(4) For any DListNode* x, if x->next == y, then y->prev == x.

(5) For any DListNode* x, if x->prev == y, then y->next == x.

(6) A DList's "size" variable is # of DListNodes, NOT COUNTING sentinel.

★Empty DList: sentinel's prev & next fields point to itself.

§ Recursion

· Two approaches to write repetitive codes: *iteration* and *recursion*.

· Recursion is usually used to solve a problem in a "divided-and-conquer" manner.

· Recursion is a repetitive process in which an algorithm calls itself.

· Use recursion when the algorithm appears within the definition itself.

· *Direct* recursion: functions that call themselves.

· *Indirect* recursion: functions that call other functions that invoke calling the function again.

◎Recursive Summation

· sum(1, n) = sum(1, n-1) + n

· sum(1, 1) = 1

```
int sum(int n)
{
    if (n==1)
        return (1);
    else
        return(sum(n-1)+n);
}
```

◎Recursive Factorial

$$\mathrm{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

⇨ recursive algorithm definition:

$$\mathrm{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (\mathrm{factorial}(n-1)) & \text{if } n > 0 \end{cases}$$

⇨ iterative factorial:

```
int iterativeFactorial(int n) {
    int i = 1;
    int factN = 1;
    while (i <= n) {
        factN = factN * i;
        i++;
    }
    return factN;
}
```

⇨ recursive factorial:

```
int recursiveFactorial(int n) {
    if (n == 0)          // base case
        return 1;
    else                 // recursive case
        return (n * recursiveFactorial(n - 1));
}
```

◎Design recursive algorithms

· Base case: the statement that "solves" the problem, e.g., factorial(0) and sum$(1, 1)$. Every recursive algorithm must have a base case.

· Recursive case: the rest of the algorithm, e.g., $n \times \mathrm{factorial}(n-1)$ and sum(1, n) = sum(1, n-1) + n. → reduce the size of the problem by recursively calling factorial and summation with $\boldsymbol{n-1}$.

· Rules for designing a recursive algorithm:

① First, determine the base case. ② Then determine the recursive (general) case.

③ Combine the base case and recursive case into an algorithm.

◎Recursive Multiplication

· a×b = a×(b-1) + a (recursive case)

· a×1 = a (base case)

```
int mult(int a, int b)
{
    if (b == 1)          // base case
        return (a);
    else                 // recursive case
        return(mult(a, b-1) + a);
}
```