Object-Oriented Programming and Data Structures

COMP2012: Abstract Data Types

Prof. Brian Mak
Prof. C. K. Tang

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Abstract Data Type

- A data structure helps store, organize, and manipulate data in a way that they can be processed efficiently by computers.
- Different applications require different data structures.
- Examples: array, linked list, (binary) tree, stack, queue, etc.
- An abstract data type (ADT) is a formal model of a data structure and is defined by its behavior from the users' perspective:
  - what operations does it support?
  - what are its possible values?

  and that is independent of its implementation. (e.g., array-based? list-based?)
- Benefits of ADTs
  - high-level of code design make it easier to understand,
  - implementation may be changed without affecting users' programs which make use of the ADTs, and
  - may be used to analyze the efficiency of algorithms.

Part I

Some ADTs

Stack and Queue

Ha, ha! I'm at the front of the QUEUE
Pick me. I am on the top of this STACK of food trays
HKUST

- Stack and queue let you insert and remove items at the ends only, not in the middle.
- Stack: *last-in first-out (LIFO)* policy.
- Queue *first-in first-out (FIFO)* policy.

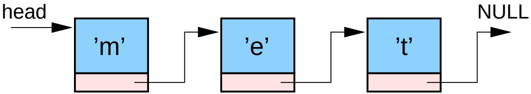
Stack ADT

```
#include <vector>                                        /* File: stack.h */
template <typename T>
class Stack
{
  public:
    Stack(void);                                     // Default CONSTRUCTOR

    bool empty(void) const;                      // Check if the stack is empty
    int size(void) const;          // Give the number of data currently stored
    T& top(void);                  // Retrieve the top item for non-const Stack
    const T& top(void) const;         // Retrieve the top item for const Stack

    void push(const T&);            // Add a new item to the top of the stack
    void pop(void);                    // Remove the top item from the stack

  private:
    std::vector<T> data;                         // Array-based implementation
    int top_index;                             // Starts from 0; -1 when empty
};
```

Queue ADT

```
#include <vector>                                        /* File: queue.h */
template <typename T>
class Queue                                                 // Circular queue
{
  public:
    Queue(void);                                   // Default CONSTRUCTOR

    bool empty(void) const;                    // Check if the queue is empty
    int size(void) const;       // Give the number of data currently stored
    T& front(void);              // Retrieve front item for non-const Queue
    const T& front(void) const;      // Retrieve front item for const Queue

    void enqueue(const T&);     // Add new item to the back of the queue
    void dequeue(void);            // Remove the front item from the queue

  private:
    std::vector<T> data;                        // Use an array to store data
    int first;                           // Index of the first item; start from 0
};
```

Linked List



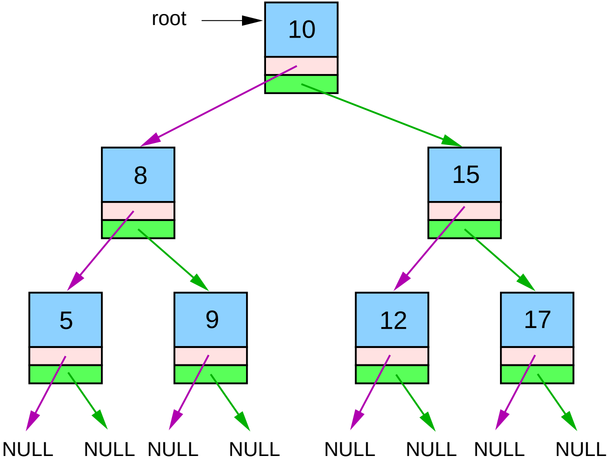
Singly Linked List ADT

```
#include <iostream>                                          /* File: sll.h */
template <typename T>
class sll {
  private:
    struct sll_node {           // Private sll_node can't be used outside sll class
        T data;                                     // Contains useful information
        sll_node* next;                             // The link to the next node
        sll_node(const T& x): data(x), next(0) { }
    };

    int size;
    sll_node* head;

  public:
    sll( ) : head(0) { }
    ~sll( );

    int size( ) const;
    bool is_empty( ) const { return head == 0; }
    void print(ostream& os = std::cout) const;

    T* ll_search(const T&) const;
    void insert(const T&);
};
```

Binary Tree

Binary Tree ..

- CS's tree actually looks like an inverted physical tree.
- In particular, any node of a binary tree has 2 sub-trees (children): left sub-tree (child) and right sub-tree (child).

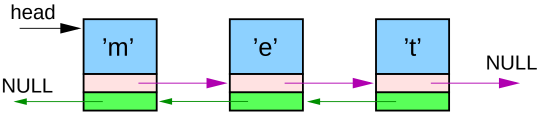


Binary Tree ADT

```
#include <iostream>                                          /* File: btree.h */
template <typename T>
class btree {
  private:
    struct btree_node {                                  // A node in a binary tree
        T data;
        btree_node* left;                       // Left sub-tree or called left child
        btree_node* right;                    // Right sub-tree or called right child
        btree_node(const T& x) : data(x), left(0), right(0) { };
    };

    btree_node* root;

  public:
    btree( ) : root(0) { }
    ~btree( );

    bool is_empty( ) const { return root == 0; }
    bool contains(const T&) const;
    void print(ostream& os = std::cout) const;

    void insert(const T&);                           // Insert an item with a policy
    void remove(const T&);                                        // Remove an item
};
```

Part II

Doubly Linked List (DLL)



Doubly Linked List (DLL)

- Doubly linked lists are useful for playing video and sound files because they allow efficient "rewind" and "instant replay".
- Lists implemented using arrays/vectors
  - arrays require the size to be known before they are created
  - element access is fast in arrays
  - insertions and deletions in the middle of arrays are inefficient
- Lists implemented using singly linked lists (SLL)
  - size of SLLs grows as needed
  - element access is slower in SLLs
  - insertions and deletions are efficient
- Now DLLs has all the same properties of SLLs except that one may move along a DLL in both forward and backward directions easily.
- In a DLL, each item points to both its predecessor and successor
  - prev: points to the predecessor
  - next: points to the successor

DLL — dll.h

```
template <typename T>                                          /* File: dll.h */
class Dll                                          // A sorted doubly linked list
{
  private:
    #include "dll-node.h"
    #include "dll-search.h"
    Dll_node* head;                                          // The first Dll_node

  public:
    Dll( ) : head(nullptr) { }    // Default constructor
    Dll(Dll_node* nodeptr) : head(nodeptr) { }   // From another list
    ~Dll( );                                // Need to release owned objects

    int size( ) const;
    void print( ) const;
    bool contains(const T& x) const;

    void insert(const T&);                              // Insert and keep sorted
    void remove(const T&);
};
```

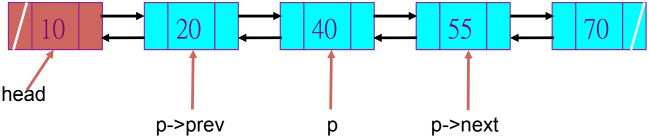
DLL — dll-node.h, dll-search.h

```
struct Dll_node                                                  /* File: dll-node.h */
{
    T data;                                               // Contains useful information
    Dll_node* prev;                                  // The link to the previous Dll_node
    Dll_node* next;                                      // The link to the next Dll_node

    Dll_node(const T& x) : data(x), prev(nullptr), next(nullptr) { };
};


// The returned pointer points to modifiable node
Dll_node* search(const T& y) const                             /* File: dll-search.h */
{
    for (Dll_node* p = head; p != nullptr; p = p→next)
    {
        if (p→data == y)
            return p;
    }
    return nullptr;
}
```

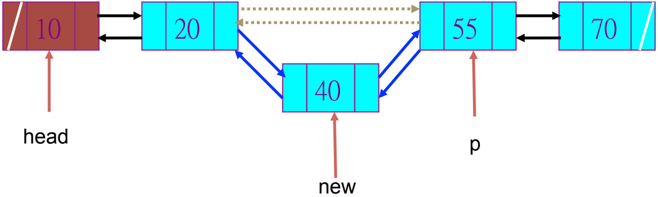
Class Dll and Private Nested Dll_node

- In this implementation, the nodes of DLL are defined as a private nested struct Dll_node. As a result,
  - the type Dll_node is unknown beyond class Dll.
  - no public member functions of Dll should call or return using Dll_node.
  - any member functions of Dll using Dll_node cannot be defined outside the Dll class.
- C++ 2011 standard adds the keyword nullptr to denote a null pointer.
- Operations of major interest:
  - insert a node
  - remove a node
  - search for a node containing a specified value
  - print (traverse) the whole DLL

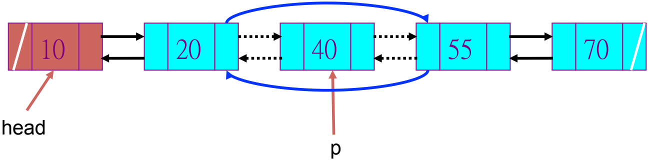
DLL Operations



Insertion of an item in the middle of a DLL



Deletion of an item in the middle of a DLL



DLL — dll-destructor.h

```
template <typename T>                              /* File: dll-destructor.h */
Dll<T>::∼Dll( )                               // To delete the WHOLE linked list
{
    if (head == nullptr)                      // An empty list; nothing to delete
        return;

    // STEP 1: Delete all nodes beyond the head by creating a
        //          temp DLL in a block. How does it work?
    {
        Dll remaining_nodes(head→next);
    }

    // For debugging only: this shows what you are deleting
    std::cout ≪ "deleting " ≪ head→data ≪ std::endl;
    delete head;                             // STEP 2: Then delete the first node

    head = nullptr;                 // STEP 3: To play safe, reset head to NULL
}
```

DLL — dll-insert.h

```
template <typename T>                                                    /* File: dll-insert.h */
void Dll<T>::insert(const T& x)
{      // Insert data in ascending order. The data type must support operator>
    Dll_node* p;
    Dll_node* new_node = new Dll_node(x);  // STEP 1: Create the new node
    // Special case: insert at the beginning
    if (head == nullptr) { head = new_node; return; }
    // STEP 2: Find the node before which the new node is to be added
    for (p = head; x > p→data && p→next != nullptr; p = p→next)
        ;
    // STEP 3: Insert the new node between the found node and its previous node
    if (x > p→data)       // Then (p->next == nullptr) and insert at the back
    {
        new_node→prev = p; p→next = new_node;
    }
    else if (p == head)                                         // Insert at the front
    {
        new_node→next = p; p→prev = new_node; head = new_node;
    }
    else                                                        // General case
    {
        new_node→next = p; new_node→prev = p→prev;
        p→prev→next = new_node; p→prev = new_node;
    }
}
```

DLL — dll-remove.h

```
template <typename T>                                   /* File: dll-remove.h */
void Dll<T>::remove(const T& x)
{
    // STEP 1: Find the item to be deleted
    Dll_node∗ p = search(x);

    if (p != nullptr)                        // Delete only if the data is found
    {                                        // STEP 2: Bypass the found item
        if (p == head)                      // Special case: delete the first item
            head = p→next;
        else
            (p→prev)→next = p→next;

        if (p→next)                          // Special case: delete the last item
            (p→next)→prev = p→prev;

        delete p;            // STEP 3: Free the memory of the deleted item
    }
    else
        std::cout ≪ "remove( ):  " ≪ x ≪ " is not there\n";
}
```

DLL — dll-utilities.h

```
template <typename T>                                    /* File: dll-utilities.h */
int Dll<T>::size( ) const
{
    int length = 0;
    for (const Dll_node* p = head; p != nullptr; p = p→next)
        ++length;
    return length;
}

template <typename T>
void Dll<T>::print( ) const
{
    for (const Dll_node* p = head; p != nullptr; p = p→next)
        std::cout ≪ p→data ≪ ' ';
    std::cout ≪ std::endl;
}

template <typename T>
bool Dll<T>::contains(const T& x) const
{
    return search(x) != nullptr;
}
```

DLL — dll-test-int.cpp

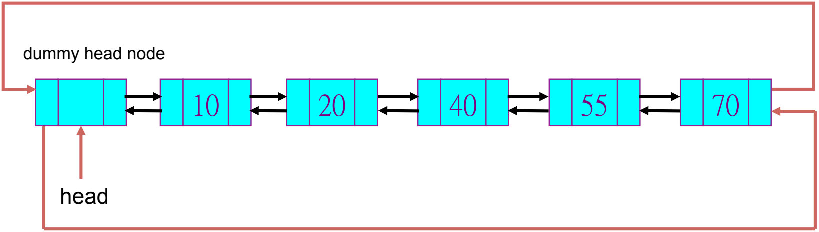
```
#include <iostream>                                    /* File: dll-test-int.cpp */
#include "dll.h"
#include "dll-destructor.h"
#include "dll-insert.h"
#include "dll-remove.h"
#include "dll-utilities.h"

int main( )
{
    Dll<int> x;
    x.insert(3); x.insert(2); x.insert(5); x.print( );
    x.insert(7); x.insert(1); x.print( );
    x.remove(0); x.remove(1); x.print( );;
    x.insert(8); x.insert(4); x.print( );
    x.remove(3); x.remove(8); x.print( );
    std::cout << "5 is " << ((x.contains(5)) ? "" : "not")
              << "in the list\n";
    return 0;
}
```

DLL — dll-test-char.cpp

```
#include <iostream>                                  /* File: dll-test-char.cpp */
#include "dll.h"
#include "dll-destructor.h"
#include "dll-utilities.h"
#include "dll-insert.h"
#include "dll-remove.h"
int main( )
{
    Dll<char> s;
    s.insert('t'); s.insert('e'); s.insert('b'); s.print( );
    std::cout << "length of s = " << s.size( ) << std::endl;
    std::cout << "delete 'e' :   "; s.remove('e'); s.print( );
    std::cout << "insert 'i' :   "; s.insert('i'); s.print( );
    std::cout << "delete 'b' :   "; s.remove('b'); s.print( );
    std::cout << "insert 'f' :   "; s.insert('f'); s.print( );
    std::cout << "delete 't' :   "; s.remove('t'); s.print( );
    std::cout << "delete 'z' :   "; s.remove('z'); s.print( );
    std::cout << "insert 'n' :   "; s.insert('n'); s.print( );
    std::cout << "insert 's' :   "; s.insert('s'); s.print( );
    return 0;
}
```

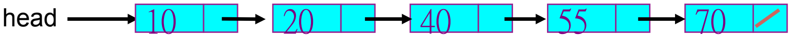
Part III

Doubly Circular Linked List With a Dummy Head Node (DHDLL)

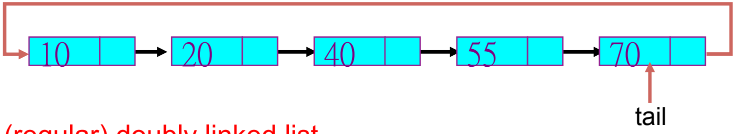


Different Kinds of Linked Lists

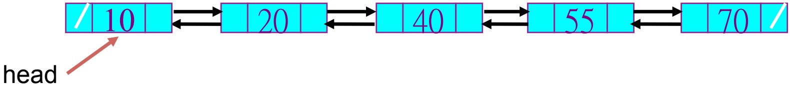
singly linked list



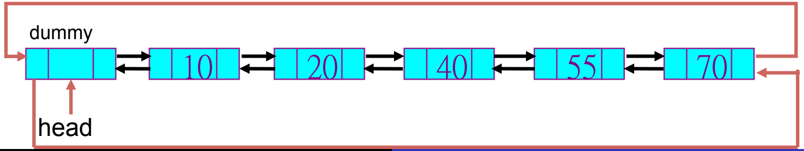
(singly) circular linked list



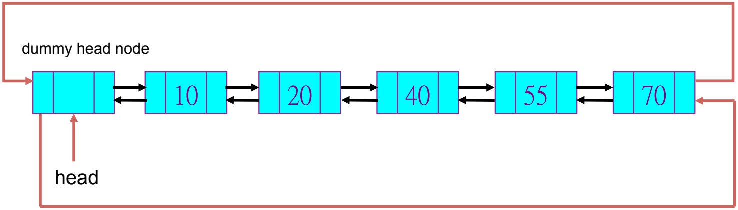
(regular) doubly linked list



doubly circular linked list with dummy head

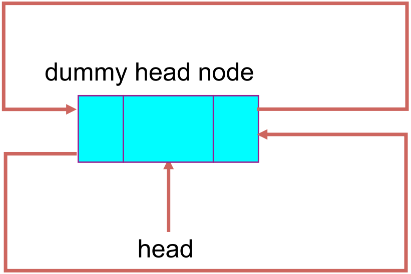


DHDLL



- Insertion or deletion of a node in a linked list generally requires different codes, depending on whether the node is
  - at the front,
  - in the middle, or
  - at the tail.
- Adding a dummy head node simplifies the code at the expense of the memory of an extra node.

Dummy Head Node and Empty DHDLL



- In general, the last node of a DHDLL points to the dummy head node (circular DHDLL).
- Don't forget to skip the dummy head node for all real operations.

DHDLL — dhdll.h

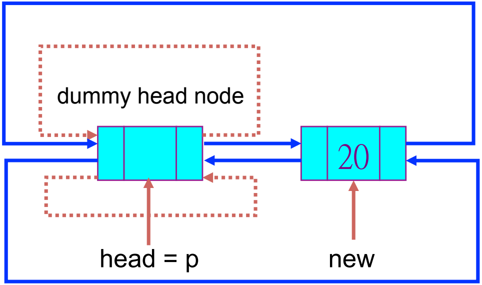
```
template <typename T>                                           /∗ File: dhdll.h ∗/
class Dhdll      // A sorted doubly circular linked list with a dummy head
{
  private:
    #include "dhdll-node.h"
    #include "dhdll-search.h"
    Dhdll_node∗ head;                                      // The first Dhdll_node

  public:
    Dhdll( ) : head(new Dhdll_node) {head→prev = head→next = head; }
    ∼Dhdll( );                                  // Need to release owned objects

    int size( ) const;
    void print( ) const;
    bool contains(const T& x) const;

    void insert(const T&);                                 // Insert and keep sorted
    void remove(const T&);
};
```

DHDLL — dhdll-node.h, dhdll-search.h

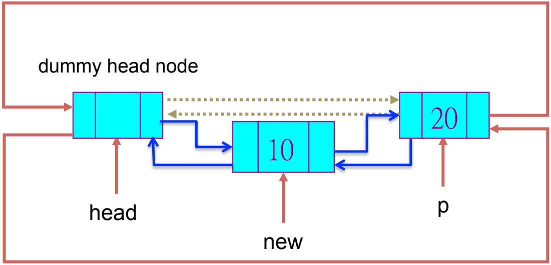
```
struct Dhdll_node                                       /∗ File: dhdll-node.h ∗/
{
    T data;                                      // Contains useful information
    Dhdll_node∗ prev;                    // The link to the previous Dhdll_node
    Dhdll_node∗ next;                        // The link to the next Dhdll_node

    Dhdll_node( ) : prev(nullptr), next(nullptr) { };
    Dhdll_node(const T& x) : data(x), prev(nullptr), next(nullptr) { };
};

// The returned pointer points to modifiable node
Dhdll_node∗ search(const T& y) const                  /∗ File: dhdll-search.h ∗/
{
    for (Dhdll_node∗ p = head→next; p != head; p = p→next)
    {
        if (p→data == y)
            return p;
    }
    return nullptr;
}
```

DHDLL Insertions

Empty:

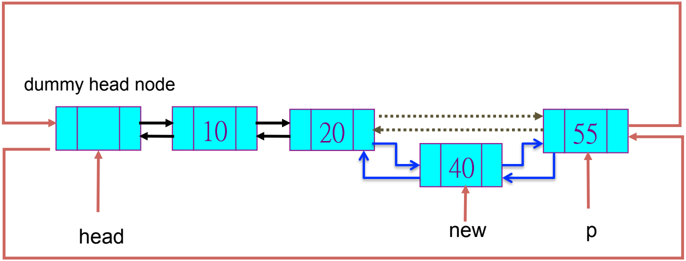


Front:

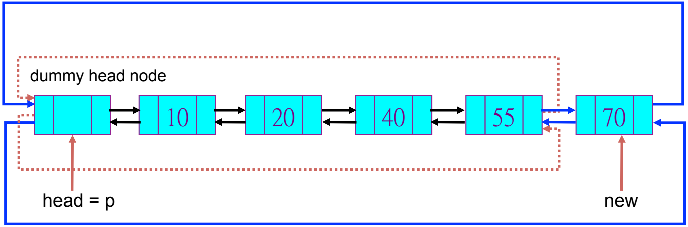


DHDLL Insertions ..

Middle:



Tail:

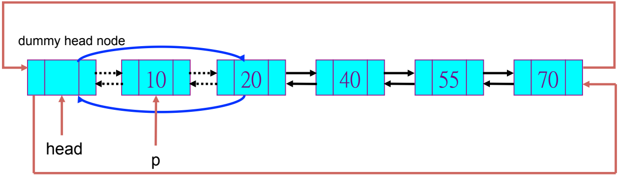


DHDLL — dhdll-insert.h

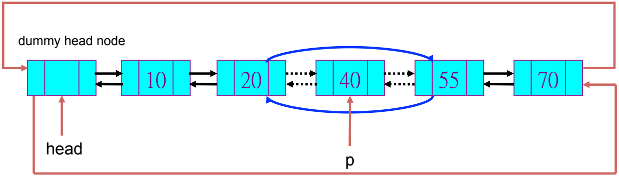
```
// Insert data in ascending order. The data type must support operator>
template <typename T>                                    /* File: dhdll-insert.h */
void Dhdll<T>::insert(const T& x)
{
    // STEP 1: Create the new node
    Dhdll_node* new_node = new Dhdll_node(x);

    // STEP 2: Find the node before which the new node will be added
    Dhdll_node* p;
    for (p = head→next; x > p→data && p != head; p = p→next)
        ;

    // STEP 3: Insert between the found node and the previous node
    new_node→next = p;                                   // Add 2 new links
    new_node→prev = p→prev;
    p→prev→next = new_node;                              // Modify 2 old links
    p→prev = new_node;
}
```

DHDLL Deletions

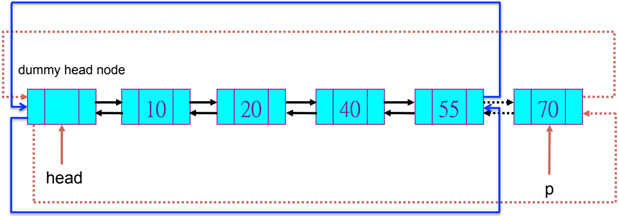
Front:



Middle:



Tail:



DHDLL — dhdll-remove.h

```
template <typename T>                                    /* File: dhdll-remove.h */
void Dhdll<T>::remove(const T& x)
{
    // STEP 1: Find the item to be deleted
    Dhdll_node* p = search(x);

    if (p != nullptr)                            // Delete only if the data is found
    {
        // STEP 2: Bypass the found item
        (p→prev)→next = p→next;                                    // Modify 2 links
        (p→next)→prev = p→prev;

        // STEP 3: Free the memory of the deleted item
        delete p;                                             // Remove 2 links as well
    }
    else
        std::cout ≪ "remove( ):  " ≪ x ≪ " is not there\n";
}
```