

# 資料結構與程式設計

## (Data Structure and Programming)

101 學年上學期複選必修課程 901 31900

Homework #5 (Due: 11:00pm, Thursday, Dec 06, 2012)

Department:\_\_\_\_\_ Grade:\_\_\_\_\_

Id:\_\_\_\_\_ Name:\_\_\_\_\_

### 0. Objectives

1. Learning how to implement various abstract data types (ADTs) for sorting input data.
2. Comparing the runtime and memory usages of various ADTs.
3. Being able to comprehend existing code and enhance/complete it.

### 1. Problem Description

In this homework, we are going to implement various ADTs including “doubly linked list”, “dynamic array” and “binary search tree”. Based on the parameter to the “*make*” command, we will generate 3 different executables: “**adtTest.dlist**”, “**adtTest.array**”, and “**adtTest.bst**”, representing the test programs for the “doubly linked list”, “dynamic array” and “binary search tree”, respectively. These generated executables share the same command interface and have the following usage (for example):

**adtTest.dlist** [-File <dofile>]

where the **bold words** indicate the command name or required entries, square brackets “[ ]” indicate optional arguments, and angle brackets “< >” indicate required arguments. Do not type the square or angle brackets.

This ADT test program should provide the following functionalities:

1. The `class AdtTest` serves as the manager for the test program. It contains an ADT (`AdtType<AdtTestObj> _container`) to store the objects of `class AdtTestObj`.
2. The `class AdtTestObj` has only 1 data member (`string _str`). Its value is specified by command (`ADTAdd -String`) or generated by random number generator (`rnGen()` in `ADTAdd -Random`). The static data member, `int _strLen`, is to confine the maximum string length for `AdtTestObj::_str`.
3. The `class AdtTestObj` objects stored in the ADT of `class AdtTest` should always be sorted in an ascending order.
4. The type of ADT is determined during compilation by the parameter for the “make” command ---
  - “make d”: defines the flag “TEST\_DLIST” so that the doubly linked list (`class DList`) will be created. Accordingly, the generated executable will be “**adtTest.dlist**”.
  - “make a”: defines the flag “TEST\_ARRAY” so that the dynamic array (`class Array`) will be created. Accordingly, the generated executable will be “**adtTest.array**”.
  - “make b”: defines the flag “TEST\_BST” so that the binary search tree (`class BSTree`) will be created. Accordingly, the generated executable will be “**adtTest.bst**”.

If none of the parameters is specified, a make error will occur.

**Note:** It will invoke “make clean” when switching between different builds.

**Note:** If you encounter unexpected compilation errors such as:

```
make[1]: *** No rule to make target
`../../include/util.h', needed by `cmdCommon.o'.  Stop.
```

Type “make clean” and then make again.

5. There should be a command to reset `class AdtTest`. The `AdtTestObj::_strLen` will be reset and the ADT in `class AdtTest` will be cleared.
6. There should be commands to add or delete objects to the ADT. The data after insertion and deletion should always be sorted in ascending order.
7. We will provide the reference codes (class, data members, member functions, and iterator class) for the doubly linked list (`class DList`) and dynamic array (`class Array`). However, you should define the binary search tree

class `BSTree` in the file “`bst.h`” from scratch. It does not need to be balanced. You don’t need to implement “rotation” for its “add” and “delete” operations. Just use straightforward methods.

**Note:** You are NOT allowed to add/remove any data member of class `DList` and class `Array`. However, feel free to define whatever data members for class `BSTree` on your own.

8. You should be able to print the elements of the ADT either in ascending or descending order (i.e. from head or from tail).
9. All the ADTs should contain the following member functions: *begin()*, *end()*, *empty()*, *size()*, *pop\_front()*, *pop\_back()*, *erase()*, *insert()*, and *clear()*.
  - *begin()*: return the iterator pointing to the first (smallest) element. Return *end()* if the ADT is empty.
  - *end()*: return the “pass-the-end” iterator. For class `DList`, *end()* is a dummy iterator whose `DListNode` has “*iterator(\_next) = begin()*”, and “*iterator(\_prev) = the last element in the list*” (i.e. forms a “ring”). If the `DList` is empty, *end()* = *begin()* = *iterator(\_head)* = the dummy iterator. For class `Array`, *end()* points to the next address of the last element. If the `Array` is NOT yet initialized (i.e. *\_capacity* == 0), both *begin()* and *end()* = 0. For class `BSTree`, you can design it on your own. But make sure the “--” operator will bring the iterator back to the last element (if ADT is not empty).
  - *bool empty()*: check whether the ADT is empty.
  - *size\_t size()*: return the number of elements in the ADT.
  - *void pop\_front()*: remove the first (smallest) element in the ADT. No action will be taken and no error will be issue if the ADT is empty. For `DList`, the `DListNode* _head` will be updated and point to its next node. For `Array`, all the elements following the popped *\_head* will be moved up front for one pointer address. However, the *\_data* pointer itself will NOT be changed. For `BSTree`, the lefttest (smallest) data should be popped out.
  - *void pop\_back()*: remove the last (largest) element in the ADT. No action will be taken and no error will be issue if the ADT is empty.
  - *bool erase(const T& x)*: remove the element *x* from the ADT. Return false if *x* does not exist in the ADT. The size of ADT, of course, will be decremented by 1 afterwards.
  - *bool erase(iterator pos)*: remove the element in the *pos* of the ADT. Return false if the ADT is empty. Otherwise, return true and we can assume that *pos* is a valid iterator in the ADT (i.e. NO need to check whether *pos* is valid or not. For example, checking whether *pos* == *end()*).

- *bool insert(const T& x)*: Insert element 'x' to the ADT. If the data "T x" has already existed in the ADT, return *false*.
  - *clear()*: empty the ADT. For `DList` and `BSTree`, delete their `DListNode` and `BSTreeNode`, respectively. Do not delete the dummy `DListNode` and `BSTreeNode` if there is one (See *end()* and Constructor). For `Array`, reset its `_size` to 0, DO NOT release its memory (i.e. `_capacity` remains the same).
10. You may also implement some private helper functions to assist the member functions above. For example, *find()*, *expand()* for class `Array`, *successor()* for class `BSTree`, etc.
11. Constructor:
- The constructor of `DList` will allocate a dummy `DListNode` for `_head` = this dummy node. Its `_prev` and `_next` are pointing to itself.
  - The constructor of `Array` will set `_data` = 0, `_size` = 0, and `_capacity` = 0. In the later data insertions, the `_capacity` will grow  $0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow \dots \rightarrow 2^n \dots$ . For data deletions, the `_capacity` will remain unchanged (i.e. Don't release memory back to system).
  - Since you should define the data members of the class `BSTree` on your own, you should also define its constructor by yourself.
12. There should be class `iterator` for each ADT with the following overloaded operators: `*li`, `++li`, `li++`, `--li`, `li--`, `=`, `!=`, and `==` (Note: "li" is an example of the iterator object). In addition, for class `Array::iterator`, `+(int)` and `+=(int)` should also be included. DO NOT overload other operators.

Please note that the command interface is completed and included in the reference code (*adtTest.h* and *adtTest.cpp* in the main/ package). Please DO NOT change them (so both files are in "MustRemove.txt"). All you need to do is to implement the various ADTs and write a test report.

Please DO NOT use the standard C library functions *memcpy()* or *memmove()* to copy/move data from one memory location to another. This is because these two functions do not call the "=" operator or copy constructor when assigning old data to the new memory location. This will cause a problem when copying/moving objects for class `AdtTestObj`.

Please note that our ADTs are not general ones. They are designed for the purpose of this homework only. Some of the common functionalities are not supported. For example, although the operator "`Array::operator []`" is defined, we cannot use it as: `a[3] = "Ric"` because this may violate the rule that all the data in the array are always sorted.

Please ignore the limitations of the ADTs as these applications will not appear in the test program.

## 2. Supported Commands

In this homework, we will support these new commands:

```
ADTReset:      (ADT test) reset ADT
ADTAdd:        (ADT test) add objects
ADTDelete:     (ADT test) delete objects
ADTPrint:      (ADT test) print ADT
```

Please refer to Homework #3 and #4 for the lexicographic notations.

### 2.1 Command “ADTReset”

Usage: **ADTReset** <(size\_t strLen)>

Description: Reset maximum string length for class `AdtTestObj` objects and clean up the ADT in class `AdtTest` (i.e. by calling `clear()` of the ADT class). The specified string length must be a positive integer.

Example:

```
adt> adtr 6    // reset maximum string length to 6
```

### 2.2 Command “ADTAdd”

Usage: **ADTAdd** <-String (string str) | -Random (size\_t repeats)>

Description: Add class `AdtTestObj` objects to the ADT. You can insert the object(s) by specified string or by random number generation. For the option “-String”, the specified string (*string str*) can contain any printable character. If the specified string, say “hello”, has already existed in the ADT, issue an error “Error: “hello” already exists!”. If the length of the specified string exceeds `AdtTestObj::_strLen`, just truncate the excessive sub-string so that its length equals to `AdtTestObj::_strLen`. Do not issue any error message. For the option “-Random”, it will generate strings in all lower-case letters and with length equal to `AdtTestObj::_strLen`. Don’t issue an error on repeated insertions for the “-Random” option.

Example:

```
adt> adta -s He11@    // insert 1 AdtTestObj whose string = “He11@”
adt> adta -r 10        // insert 10 AdtTestObj’s by random number generation
```

## 2.3 Command “ADTDelete”

Usage: **ADTDelete** < **-All** | **-String** (string str) |  
< < **-MINimal** | **-MAXimal** | **-Random**> (size\_t repeats) > >

Description: Delete objects from the ADT. You can delete the entire ADT (*-all*), a specific string (*-String*), the minimal or maximal item, or some random data in the ADT. If the specified string (*-String*), say “hello”, is not found, issue an error “Error: “hello” is not found!”. Don’t issue errors for *-All*, *-Min*, *-Max*, or *-Random* options, even if the number of elements in the ADT is smaller than the specified times of deletions, or the string-to-be-deleted is not in the ADT (for *-Random*).

Examples:

```
adt> adtd -all          // delete all elements; the ADT becomes empty afterwards
adt> adtd -s kkk        // delete the element with string = “kkk”
adt> adtd -r 3           // randomly delete 3 elements; do not check repeats.
adt> adtd -min 3         // delete three smallest elements
```

## 2.4 Command “ADTPRrint [-Reversed]”

Usage: **ADTPrint** [-Reversed]

Description: print out the elements in the ADT. By default, print out the numbers in ascending order. If the option “-Reverse” is specified, print in descending (reversed) order.

Example:

```
adt> adtp              // print the ADT in ascending order
adt> adtp -r           // print the ADT in descending order
```

## 3. What you should do?

You are encouraged to follow the steps below for this homework assignment:

1. Read the specification carefully and make sure you understand the requirements.
2. Think first how you are going to write the program, assuming you don’t have the reference code.
3. Study the provided source code. Please note that the “cmd” package has been precompiled as “libcmd-32.a” and “libcmd-64.a” for 32 and 64-bit platforms, respectively. Use “**make 32**” or “**make 64**” to change the symbolic links in directory “lib” to suit your platform. Note that we will not test special keys in

this homework. However, if you have different keyboard mapping and would like to use the special keys, please go ahead to copy your own “*cmd*” package to “*src*” and modify the “REFPKGS” and “SRCPKGS” macros in Makefile accordingly.

4. The classes and interface functions for ADT\* commands are included in files *adtTest.h* and *adtTest.cpp* under the *main* directory. You don’t need to work on the command interface in this homework.
5. Implement the member functions and overloaded operators for classes *DList*, *Array* and their iterators.
6. Work on the “almost empty” header file “*bst.h*” in directory “*util*” and implement classes *BSTree*, *BSTreeNode* and its iterator class for the binary search tree ADT. Please note that this item is quite challenging. Just do your best and don’t get frustrated if you cannot finish it.
7. Complete your coding and compile with “make d”, “make a”, or “make b”. You should see 3 different executables. Test your programs thoroughly.
8. Some test scripts are available under the “*tests*” directory. Another script “*do.all*” in hw5 root directory allows you to run through all the scripts (e.g. “*do.all adtTest.array*”)
9. Design testcases (dofiles) to compare the performance of the doubly linked list, dynamic array, and binary search tree under different operations. You should use your creativity to construct different scenarios to compare the runtime and memory usage (Note: use the “*usage*” command) of various ADTs. Please write down a report, convert it to a PDF file named “*adtComp.pdf*”, and include it into your “.tgz” file.

## 4. Grading

We will test your submitted programs with various combinations/sequences of commands to determine your grade. The results (i.e. outputs) will be compared with our reference programs. Minor differences due to printing alignment, spacing, error message, etc can be tolerated. However, to assist TAs for easier grading work, please try to match your output with ours. Our grading will focus on the correctness and efficiency of your program, and on the performance study report “*adtComp.pdf*”.

## 5. Notes

1. Ideally, the values (strings) in the `AdtTestObj` objects should match our reference program since we use the same random number generator with the same random seed. However, if your code performs some extra copies or constructions on the `AdtTestObj` objects (which are temporary objects and will not be recorded in the ADT), you may see different numbers between your program and the reference program. Try to fix this as much as you can in order to save our efforts in grading your homework. However, this may not be easy due to different implementations. So we will ---

Use “*-String*” to test correctness, and use “*-Random*” to test performance.

2. There's a hidden option “*-Verbose*” in command “ADTPrint” for the “adtTest-ref.bst” reference program. It will print out the binary search tree on the screen in addition to the ADT content.
3. Once again, BST is not easy. There are many things you need to understand and consider ---

- (i) Do we need “`BSTreeNode<T> *BSTree::_tail`”? Why should we need it? Pointed to a dummy node? Anyway, it should be updated in “insert” and “erase”
- (ii) Do we need “`BSTreeNode<T>* BSTreeNode<T>::_parent`”? Why should we need it? When inserting/erasing a node, needs to update parent's (`_left`, `_right`) pointer. When deleting `min()`/successor() node, needs to update parent's (`_left`, `_right`) pointer. Can we do without it? What's the trade-off
- (iii) What does “`iterator BSTree<T>::begin()`” refer to? Return `iterator(_root)`? No!! “`begin()`” is supposed to point to the smallest element. In addition, we may need to update it after “insert” and “erase”.
- (iv) What does “`iterator& iterator::operator ++()`” do? Who's next? How to get to the next iterator? Recursive vs. iterative styles of tree traversal code. How about `operator --()`?

4. Notes about my implementation of BST. But you DON'T need to follow me!!

- (i) 最後還是決定不要用 `_tail` (dummy node) 了, 因為好像沒什麼必要, 但卻會引起一個不太好處理的 bug...



- (ii) 沒有 `_parent`, 原因是覺得 `maintain` 他太麻煩
- (iii) 因為沒有 `_parent`, 所以當一個 `node` 的 `right = 0` 時, `successor()` 並不好找... 所以 `keep` 了一個 `_trace` 在 `iterator` 裡, 記錄這個 `iterator` 走過的痕跡... 不過我不是用 `static data member`, 因為這樣無法支援多個 `iterator` 同時存在, 而且 `li++` 及 `li--` 也沒有辦法同時使用. 至於我的 `trace` 怎麼做... 其實還蠻簡單的, 應該是少於 50 行 `code`, 所以我賣個關子...
- (iv) `Debug` 心得: 如果不是 `runtime crash` 而是 `logical error` 的話, 建議也學 `reference program` 那樣寫一個 `print` 的 `function`, 將 `BST` 印出來, 看看有沒有 `insert or delete` 錯誤.

## 5. 關於 `Reference code` 不用 `_parent` 的寫法:

簡單的說明一下, 但是我這樣的作法是有點 `tricky`, 你如果已有你自己的作法, 其實真的可以不用參考我的。

首先, 我的 `BSTree::iterator` 裡存了一個 `trace`, 記錄這個 `iterator` 目前到此為止走過的 `trace`, 而所謂的 `trace`, 就是 `(node, left/right)...` 的 `sequence`。至於細節, 恕不奉告 (也許以後有機會再跟大家分享)。

一些需要考慮的因素:

- (i) 這個 `_trace data member` 是個 `object`, 不是 `pointer`, 所以每個 `iterator` 會有自己的 `_trace`。
- (ii) 當 `iterator copy` 的時候 (如 `lj = li`, 或是 `li++`), `trace` 也會 `copy` 過去。`destructor` 當然也會自動將 `trace` 清掉 (i.e. call its `destructor`)。
- (iii) 那 `trace` 的頭是什麼? 就是 `construct "iterator(n)"` 的那個 `n`, 在我的 `implementation` 裡, 他只會是 `_root` (因為其他的 `node` is not accessible to the users)。
- (iv) `++/--` 不只是一要 `push/pop` 一個 `trace`, 還要考慮 往上走在往下走的情形, 不過有 `_trace` 的話這個還蠻簡單的, 大家可以想想看。
- (v) 走到底的話, `++/--` 就不要走了。

## 6. 關於 `Performance study report` “`adtComp.pdf`” 需要寫的內容如下:

### 一、資料結構的實做

1. 簡述關於這三種 `ADT` 的資料結構與操作上之不同不需描述太多, 有“系統”的“清楚”“簡單”說明就好, 切勿直接貼上網路參考資料。

2. 你如何去使用程式去實現上述的概念簡單說明“程式”的實做方式，實做上關鍵的部份，切勿直接貼上 **CODE**。若貼上 **pseudo code** 也請再用“文字說明”。
3. 請說明為何你要使用這樣的實做方式，有何優缺點。如同上項 **Note #4** 內的說明，簡述一下為何你要這樣實做。  
像是，1. 好寫 2. **performance** 好 3. 其他理由...

## 二、實驗比較，請說明你所設計的實驗以及比較這些 **ADT** 的 **Performance**

1. 實驗設計
2. 實驗預期
3. 結果比較與討論

報告有繳交沒有亂寫就有基本分。但如果報告沒交或是亂寫是會 0 分的！！請注意！！

至於報告著重質量不重數量。但是有質量且有數量也是值得鼓勵的，只是分數會 **saturated**。

基本上將關鍵字組合成完整句子並且有調理的整理且簡單說明，即可。