資料結構與程式設計

(Data Structure and Programming)

107 學年上學期複選必修課程 901 31900

Homework #5 (Due: 9:00pm, Tuesday, Nov 19, 2019)

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 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 "**adTst.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.
   ```

   Please type "*make d|a|b*" again accordingly.

4. There should be a command to reset `class AdtTest`. The AdtTestObj::_strLen will be reset and the ADT in `class AdtTest` will be cleared.

5. There should be commands to add or delete objects to the ADT.

6. There should be a command to sort the `class AdtTestObj` objects stored in `class DList` and `class Array`. Note that you don't need to maintain the order of the data for other commands. For `class DList`, please sort the data using the data structure itself (i.e. `DList`), DO NOT copy the data to other data structure (e.g. `Array`) just to lower the computational complexity. Ideally, the space complexity (i.e. extra memory required) for sorting should be O(1). Data in `class BSTree` by definition should always be sorted.

7. There is also a data member *bool _isSorted* for these two classes. You can then optionally use it to indicate whether the data is in a sorted state and thus take advantage of it to find a specific data whenever applicable. However, there is no command to report the value of this data member. Therefore, you

can design your own mechanism to maintain its value for your own optimization purpose (and of course, you can also choose NOT to use it).

8. 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: To conform to regular ADT implementation, 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.

9. You should be able to find an element in the ADT, and print the elements of the ADT either in forward or backward order.

10. All the ADTs should contain the following member functions: *begin(), end(), empty(), size(), pop_front(), pop_back(), erase(), find(),* and *clear()*.
    - *iterator begin()*: return the iterator pointing to the first element. For `BSTree`, it is the leftmost element. Return *end()* if the ADT is empty.
    - *iterator end()*: return the "past-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 element in the ADT. No action will be taken and no error will be issue if the ADT is empty. If not empty, for `DList`, the `DListNode*` *_head* will be updated and point to its next node. For `Array`, the first element will be removed and, if *_size* >= 2, the last element will be "copied" to the first position. However, the *_data* pointer itself and *_capacity* will NOT be changed (i.e. the *popFront()* for `Array` should have O(1) complexity). For `BSTree`, its leftmost node (i.e. *begin()*) will be removed.
    - *void pop_back()*: remove the last (rightmost for `BSTree`) 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. If there are multiple existences of element *x* in the ADT, remove the firstly encountered one (i.e. by traversing from *begin()*) and leave the others untouched. 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, no need to check whether pos == end()).
- iterator find(const T& x): check whether the element 'x' is in the ADT. If yes, return the iterator where 'x' is found. Otherwise, return "*end()*". Please note that the time complexity for find() varies among different data structures. However, the space complexity for find() should always be O(1).
- *void clear()*: empty the ADT. For `DList` and `BSTree`, delete all of 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).

11. The member functions to add a new data to ADTs are different between `Array`, `DList` and `BSTree`. For `Array` and `DList`, new data is added to the end of the ADT by the *push_back(const T& x)* function, and for `BSTree`, new data is added by the *insert(const T& x)* function in order to maintain the relative order of the stored data. Please note that duplicated data is allowed in all ADTs. That is, both *push_back()* and *insert()* functions should have return type *void*.

12. You may also implement some private helper functions to assist the member functions above. For example, *findRecur()*, *expand()* for `class Array`, and *successor()* for `class BSTree`, etc.

13. 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 → 1 → 2 → 4 →... → 2^n... For data deletions, the *_capacity* will remain unchanged (i.e. Don't release memory back to system).
    - You should decide the initial value for the data member *_isSorted*. You can even ignore it if you don't use it.
    - Since you should define the data members of the `class BSTree` on your own, you should also define its constructor by yourself.

14. There should be inner `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`.

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
ADTQuery:     (ADT test) Query if an object exists
ADTSort:      (ADT test) sort ADT
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 8    // reset maximum string length to 8
```

2.2 Command "ADTAdd"

Usage: **ADTAdd** <**-String** (string str) | **-Random** (size_t repeats)>

Description: Add `class AdtTestObj` objects to the ADT. You can add 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. For `Array` and `DList`, the specified string is added to the end of ADT. For `BSTree`, the string should be inserted to the proper position of the sorted ADT. We do not check whether the ADT has already contained the specified string, and in such case, duplicated data will be added to the ADT. 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** (stirng str) |

< < **-Front** | **-Back** | **-Random**> (size_t repeats) > >

Description: Delete objects from the ADT. You can delete the entire ADT (-*all*), a specific string (-*String*), from the first (-*Front*) or from the last (-*Back*) 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!`". If there are multiple elements with the specified string (-*String*), delete the first encounter and leave the other(s) alone. Don't issue errors for -*All*, -*Front*, -*Back*, or –*Random* options, even if the number of elements in the ADT is smaller than the specified times of deletions.

Examples:

```
adt> adtd -all    // delete all elements; ADT becomes empty afterwards
adt> adtd -s kkk  // delete the first element with string = "kkk"
adt> adtd -r 3     // randomly delete 3 elements; do not check repeats.
// Note that for class Array,
// when deleting the first element, the last element will replace its position.
// So assume current ADT is an array and its content is: { a, b, c, d, e, f, g }
adt> adtd -f 1   // delete the first element; array becomes: { g,b,c,d,e,f }
adt> adtd -f 3   // delete the first element; repeated for 3 times
                 // array now becomes: { d, b, c }, NOT: { d, e, f }
```

2.4 Command "ADTQuery"

Usage: **ADTQuery** <(stirng str)>

Description: Query if there is an object in the ADT with the data "str". The length of "str" should be equal to or smaller than the *AdtTestObj::_strLen*. If exceeds, print out an error message.

Examples:

```
adt> adtq yyy    // query whether "yyy" exists
"yyy" is found.
adt> adtdq zzz   // query whether "zzz" exists
"zzz" is not found!!
adt> adtq fsalkgjsalgk
Error: "fsalkgjsalgk" exceeds string length limit!!
```

2.5 Command "ADTSort"

Usage: **ADTSort**

Description: Sort the objects in ascending order. Issue an error if any argument is provided. For `BSTree`, this is a dummy command (i.e. no action will be taken) since all the data have already been sorted. Optionally, you can use the data member "_isSorted" to check whether sorting is necessary.

2.6 Command "ADTPrint"

Usage: **ADTPrint** [**-Reversed | (int n)**]

Description: Print out the elements in the ADT. By default, print out the entire ADT in forward order. If the option "-Reverse" is specified, print it in backward (reversed) order. If a number "n" is specified, print out the n^{th} element in the ADT, i.e. for linked list and array, it is the n^{th} element in line (indexing starts from '0'); for binary search tree, it is the n^{th} smallest element in the ADT (n=0 is the smallest). If n is not an integer, smaller than 0, or greater than or equal to the size of the ADT, print out an error message.

Example:

```
adt> adtp        // print the ADT in forward order
adt> adtp -r     // print the ADT in backward order
adt> adtp 13     // print the 13th element in the ADT
[ 13] = csoxla
adt> adtp -1
Error: "-1" is not a legal index!!
adt> adtp 12345678
```

```
Error: "12345678" is not a legal index!!
adt> adtp ric
Error: Illegal option!! (ric)
```

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-linux.a*" and "*libcmd-mac.a*" for Linux and Mac platforms, respectively. They are as the same from Homework #4. Use "**make linux**" or "**make mac**" in root directory to change the symbolic links in the 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 and modify the "REFPKGS" and "SRCPKGS" macros in Makefile accordingly. We will restore it when testing your program.

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 strings 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.

   Note that reference program was compiled with "-O3" flag, which may run a few times faster than those with "-g" flag. Switch the line "CFLAGS" in "src/Makefile.in" if you want to test your program with "-O3".

2. Note that the `Array::sort()` function is provided. We just call the global *::sort()* function from STL. However, for `DList`, the *std::sort()* function won't work since it takes `RandomAccessIterator`. You should implement the sorting function on your own. Please do not copy the data of `DList` to other types of ADT (e.g. `Array` or `vector`) to perform sorting. The performance of `DList::sort()` is expected to be $O(n^2)$ for time and O(1) for space. For `BSTree`, the *sort()* function is empty as its data has already been sorted when inserted.

3. 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.

4. 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? If you use it, please note that 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 --()?

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

(i) I decided NOT to keep "_tail" (dummy node). It's mainly because "_tail" is not required in my algorithm and yet it will create some tricky bugs if I use it.

(ii) No "_parent", because I think it's too complicated to maintain it.

(iii) Since I dom't keep the"_parent" information, it's a bit tricky when I need to find the successor if the right child is NULL. Therefore, I will a "_trace" in my "iterator" to record the how it is traversed from the "_root". However, "_trace" is NOT a static data member so that I can support multiple co-existences of iterators (e.g. using li++ and li—at the same time). As for how I implement "_trace", well, it is not that complicated (please see '6' below). The number of lines of codes for "_trace" is less than 50, FYI.

(iv) What I learned from debugging BST: If it is not a runtime crash but a logical error, try to implement a "verbose print" function as reference program. It helps visualize BST especially in debugging "insert" or "delete".

6. How I implemented BST without "_parent":

Although it's not that complicated, it's a bit tricky. You don't need to follow mine. You should design it on your own. This is just FYI.

Firstly, I store a "_trace" in BSTree::iterator to record the trace of how it is traversed from "_root". A record in a trace is nothing but a (node, left/right)

pair, where “node” is the “from node”, and “left/right” indicates whether it is traversed from node’s left or right child.

Some factors to consider:

(i) Data member “_trace” is an object, not a pointer, so every iterator has its own “_trace”.

(ii) When copying an iterator (e.g. lj = li, or li++), “_trace” will be copied too. Destructure will clear “_trace” (i.e. call its destructor).

(iii) Then what’s the head of “_trace”? It’s the ‘n’ in constructor “iterator(n)”. In my implementation, it can only be “_root” because other nodes are NOT accessible to the users.

(iv) ++/-- is NOT just pushing/popping a trace node. It should also consider when it traverses up and down. However, with “_trace”, this is quire straightforward. You can think about it.

(v) When it reaches the rightmost/leftmost code, no more traversal is possible for ++/--.

7. Some reminders from TA:
   (i) SelfCheck 只能幫你檢查檔名，這次因為報告要交 pdf，麻煩自己確認交上來的是打得開的格式，以往會有人交上來是亂碼或者交名為 adtComp.pdf 的資料夾。另外，這次作業雖然是三個不同的資料結構，但我在收作業的時候不接受部分補交(就全部視為同一份)，總共四個檔案(3 份程式+1 份報告)在交上來前請多加檢查確認。
   (ii) 整體難度大概是 bst >> dlist >= array，但三者配分一樣，建議從後兩者開始寫，如果你 bst 真的寫不出來，也麻煩盡量交個編譯得過的版本上來。
   (iii) 程式部分以下兩點最重要，請務必先完成
      ✓ 插入(adta)：其他的功能幾乎是建立在已經插入資料後，所以測試其他功能前會先插入資料
      ✓ 輸出(adtp, adtp -r)：我們會同時檢查正向(begin 到 end)以及反向(end 到 begin)來確保你的結構連接完全正確
   (iv) 與 ref program 有一點效能的差異不會被扣分，但如果真的差很多(如：開 O3 之後還是差十倍以上)，可能會被 killed 而得到 0 分。所以如果你的 code 比 ref 慢不少，可能要檢查是不是有哪裡寫得不好，或甚至寫錯了。比 ref program 快不會加分 ☺
   (v) 報告部分，實作與實驗各佔一半，實作簡單說明即可，不需要貼一堆程式碼，也可以補充設計架構的想法或心得，實驗部分則建議用表格或畫圖來呈現結果，可以比較明顯看出各結構間的差異。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。總之，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。