Unit 6 Dictionaries

College of Computer Science, CQU

outline

□ The ADT for a simple dictionary

□ Example:A payroll record implementation

□ A dictionary search example

□ Implementation for a class representing a key-value pair

□ A dictionary implemented with an unsorted array-based list

□ Dictionary implementation using a sorted array-based list

dictionary

- **Dictionary is a collection of data records, an** efficient ways to organize collections of data records so that they can be stored and retrieved quickly.

- Key: a number which can be used for searching records

- comparable

The ADT for a simple dictionary

- // The Dictionary abstract class.
- template <typename Key, typename E>
- class Dictionary {
- private:
- void operator =(const Dictionary&) {}
- Dictionary(const Dictionary&) {}
- public:
- Dictionary() {} // Default constructor
- virtual ˜Dictionary() {} // Base destructor
- // Reinitialize dictionary
- virtual void clear() = 0;

The ADT for a simple dictionary

- // Insert a record
- // k: The key for the record being inserted.
- // e: The record being inserted.
- **virtual void insert(const Key& k, const E& e) = 0;**
- // Remove and return a record.
- // k: The key of the record to be removed.
- // Return: A maching record. If multiple records match
- // "k", remove an arbitrary one. Return NULL if no record
- // with key "k" exists.
- **virtual E remove(const Key& k) = 0;**
- // Remove and return an arbitrary record from dictionary.
- // Return: The record removed, or NULL if none exists.
- **virtual E removeAny() = 0;**

The ADT for a simple dictionary

- **// Return: A record matching "k" (NULL if none exists).**

- **// If multiple records match, return an arbitrary one.**

- **// k: The key of the record to find**

- **virtual E find(const Key& k) const = 0;**

- **// Return the number of records in the dictionary.**

- **virtual int size() = 0;**

- **};**

Example: A payroll record implementation

```cpp
// A simple payroll entry with ID, name, address fields
class Payroll {
private:
    int ID;
    string name;
    string address;
public:
// Constructor
Payroll(int inID, string inname, string inaddr) {
ID = inID;
name = inname;
address = inaddr;
}
~Payroll() {} // Destructor
// Local data member access functions
int getID() { return ID; }
string getname() { return name; }
string getaddr() { return address; }
};
```

Example: a dictionary search example of payroll record

```
int main() {
// IDdict organizes Payroll records by ID
UALdict<int, Payroll*> IDdict;
// namedict organizes Payroll records by name
UALdict<string, Payroll*> namedict;
Payroll *foo1, *foo2, *findfoo1, *findfoo2;
foo1 = new Payroll(5, "Joe", "Anytown");
foo2 = new Payroll(10, "John", "Mytown");
IDdict.insert(foo1->getID(), foo1);
IDdict.insert(foo2->getID(), foo2);
namedict.insert(foo1->getname(), foo1);
namedict.insert(foo2->getname(), foo2);
findfoo1 = IDdict.find(5);
if (findfoo1 != NULL) cout << findfoo1;
else cout << "NULL ";
findfoo2 = namedict.find("John");
if (findfoo2 != NULL) cout << findfoo2;
else cout << "NULL ";
}
```

Here, payroll records are stored in two dictionaries, one organized by ID and the other organized by name. Both dictionaries are implemented with an unsorted array-based list.

Mechanism for extracting keys

- The **fundamental operation** for a dictionary is finding a record that matches a given key.

- This raises the issue of how to extract the key from a record.

- We would like any given dictionary implementation to support arbitrary record types, so we need some mechanism for extracting keys that is sufficiently general.

Mechanism for extracting keys

- **One approach:** to require all record types to support some particular method that returns the key value.

- **Problem:** this approach does not work when the same record type is meant to be stored in multiple dictionaries, each keyed by a different field of the record.

Mechanism for extracting keys

- **Another approach:** to supply a class whose job is to extract the key from the record.

- **Problem:** this solution also does not work in all situations, because there are record types for which it is not possible to write a key extraction method.

- The fundamental issue is that the key value for a record is not an intrinsic property of the record's class, or of any field within the class. The key for a record is actually a property of the context in which the record is used.

Mechanism for extracting keys

- **Solution: key-value pairs**

- **to explicitly store the key associated with a given**

record, as a separate field in the dictionary.

- **That is, each entry in the dictionary will contain both a record and its associated key. Such entries are known as key-value pairs.**

Implementation for a class representing a key-value pair

- // Container for a key-value pair
- template <typename Key, typename E>
- class KVpair {
- private:
-     Key k;
-     E e;
- public:
- // Constructors
- KVpair() {}
- KVpair(Key kval, E eval)
- { k = kval; e = eval; }
- KVpair(const KVpair& o) // Copy constructor
- { k = o.k; e = o.e; }
- void operator =(const KVpair& o) // Assignment operator
- { k = o.k; e = o.e; }
- // Data member access functions
- Key key() { return k; }
- void setKey(Key ink) { k = ink; }
- E value() { return e; }
- };

Ways to implement dictionary :implemented with an unsorted array-based list

- // Dictionary implemented with an unsorted array-based list
- template <typename Key, typename E>
- class UALdict : public Dictionary<Key, E> {
- private:
- AList<KVpair<Key,E> >* list;
- public:
- UALdict(int size=defaultSize) // Constructor
- { list = new AList<KVpair<Key,E> >(size); }
- ˜UALdict() { delete list; } // Destructor
- void clear() { list->clear(); } // Reinitialize
- // Insert an element: append to list
- void insert(const Key&k, const E& e) {
- KVpair<Key,E> temp(k, e);
- list->append(temp);
- }

A dictionary implemented with an unsorted array-based list

- // Find "k" using sequential search
- **E find(const Key& k) const {**
- for(list->moveToStart();
-     list->currPos() < list->length(); list->next()) {
-         KVpair<Key,E> temp = list->getValue();
-         if (k == temp.key())
-             return temp.value();
-     }
-     return NULL; // "k" does not appear in dictionary
- }
- **int size()** // Return list size
- { return list->length(); }
- };

A dictionary implemented with an unsorted array-based list

- // Use sequential search to find the element to remove

- E remove(const Key& k) {

- E temp = find(k); // "find" will set list position

- if(temp != NULL) list->remove();

- return temp;

- }

- E removeAny() { // Remove the last element

- Assert(size() != 0, "Dictionary is empty");

- list->moveToEnd();

- list->prev();

- KVpair<Key,E> e = list->remove();

- return e.value();

- }

An implementation for a sorted array-based list: Inherit from AList

- // Sorted array-based list
- // Inherit from AList as a protected base class
- template <typename Key, typename E>
- class SAList: protected AList<KVpair<Key,E> > {
- public:
- SAList(int size=defaultSize) :AList<KVpair<Key,E> >(size) {}
- ˜SAList() {} // Destructor
- // Redefine insert function to keep values sorted
- void insert(KVpair<Key,E>& it) { // Insert at right
-   KVpair<Key,E> curr;
-   for (moveToStart(); currPos() < length(); next()) {
-     curr = getValue();
-     if(curr.key() > it.key())   break;
-   }
-     AList<KVpair<Key,E> >::insert(it); // Do AList insert
-   }

An implementation for a sorted array-based list: Inherit from AList

- // With the exception of append, all remaining methods are
- // exposed from AList. Append is not available to SAlist
- // class users since it has not been explicitly exposed.
- AList<KVpair<Key,E> >::clear;
- AList<KVpair<Key,E> >::remove;
- AList<KVpair<Key,E> >::moveToStart;
- AList<KVpair<Key,E> >::moveToEnd;
- AList<KVpair<Key,E> >::prev;
- AList<KVpair<Key,E> >::next;
- AList<KVpair<Key,E> >::length;
- AList<KVpair<Key,E> >::currPos;
- AList<KVpair<Key,E> >::moveToPos;
- AList<KVpair<Key,E> >::getValue;
- };

A dictionary implemented with a sorted array-based list

- // Dictionary implemented with a sorted array-based list
- template <typename Key, typename E>
- class SALdict : public Dictionary<Key, E> {
- private:
-   SAList<Key,E>* list;
- public:
-   SALdict(int size=defaultSize) // Constructor
-      { list = new SAList<Key,E>(size); }
-   ˜SALdict() { delete list; } // Destructor
-   void clear() { list->clear(); } // Reinitialize
-     // Insert an element: Keep elements sorted
-     void insert(const Key&k, const E& e) {
-        KVpair<Key,E> temp(k, e);
-        list->insert(temp);
-     }

A dictionary implemented with a sorted array-based list

- □   // Use sequential search to find the element to remove
- □   **E remove(const Key& k) {**
- □   **E temp = find(k);**
- □   **if (temp != NULL) list->remove();**
- □   **return temp;**
- □   **}**
- □   **E removeAny() { // Remove the last element**
- □   **Assert(size() != 0, "Dictionary is empty");**
- □   **list->moveToEnd();**
- □   **list->prev();**
- □   **KVpair<Key,E> e = list->remove();**
- □   **return e.value();**
- □   **}**

A dictionary implemented with a sorted array-based list

```
// Find "K" using binary search
E find(const Key& k) const {
    int l = -1;
    int r = list->length();
    while (l+1 != r) { // Stop when l and r meet
        int i = (l+r)/2; // Check middle of remaining subarray
        list->moveToPos(i);
        KVpair<Key,E> temp = list->getValue();
        if (k < temp.key()) r = i; // In left
        if (k == temp.key()) return temp.value(); // Found it
        if (k > temp.key()) l = i; // In right
    }
    return NULL; // "k" does not appear in dictionary
}
```

Reference

- **P134----P145**

-End-