Pool: An Unbounded Array*

Pei-Chi Wu Feng-Jian Wang

Institute of Computer Science and Information Engineering
National Chiao Tung University
1001 Ta-Hsueh Road, Hsinchu
Taiwan, Republic of China
{pcwu, fjwang}@csie.nctu.edu.tw

ABSTRACT

Many collection objects, e.g., stack, can be implemented with tabular representation (an array). Such an implementation provides fast index and saves space on linked structures. However, its space is bounded at creation time. This note presents a hybrid data structure, called *pool*, which combines tabular and indexed data structures. A pool is a number of segments of contiguous memory space. It provides unbounded space. Its first segment is allocated initially. When more space is needed, a new segment is allocated in double size of previous segment. All segments allocated can be accessed via an index table. The size of index table is log n, where n is the number of elements. A pool can provide index operation in $O(\log n)$ and iterations almost as fast as tabular implementation.

Keywords: dynamic data structures, collection objects.

1. MOTIVATION

A collection object (e.g., a stack, a queue) is a dynamic data structure whose size cannot be determined at the creation time. Its implementation usually contains many pointers that link its elements. There is some space overhead for storing these links and some time overhead when accessing data indirectly. Tabular representation [3] is an implementation technique that represents a collection object in a contiguous memory space (an array). The tabular representation is suitable for a collection object that needs to maintain some "linear" relationship between its elements. The linear relationship is not implemented by additional links but by physical adjacency of elements.

Such a technique has been widely used (although may not be explicitly addressed). For example, a stack can be represented as an array of elements and a variable `count` as the number of elements in the stack. The variable `count` is increased when an element is pushed, and it is decreased when an element is popped. There are no additional links needed, in comparison with a linked list implementation. The tabular representation is fast since no links need be set when pushing or popping an element. When the size of a collection object is fixed, it can also save space.

Some data movements may be needed for collection objects in tabular representation. A (bounded) contiguous space is first allocated in the creation time of a collection object. When the size of the collection object exceeds the predetermined size, its contents are copied to a new and larger memory space. The overall cost (in worst case) is $O(n^2)$, when the space allocated is linearly increasing each time.

---

* This research was partly supported by National Science Council, Taiwan, R.O.C., under Contract No. NSC 82-0408-E009-280.

2. POOL

This section presents a hybrid data structure, called *pool*, which combines tabular and indexed data structures. A pool is a number of segments of contiguous memory space. It provides unbounded space. The first segment is allocated initially. Its size is *16* or more. When more space is needed, the size of a new allocated segment is the double of current segment. All segments allocated can be accessed via an index table. Since the size of segment increases exponentially, the size of index table is log n, where n is the number of elements in a collection. For a 32-bit machine, the size of index table is smaller than 32. The index table can then be implemented as an array. This implementation can provide index operation in $O(\log n)$ time and the iterators as fast as tabular implementation.

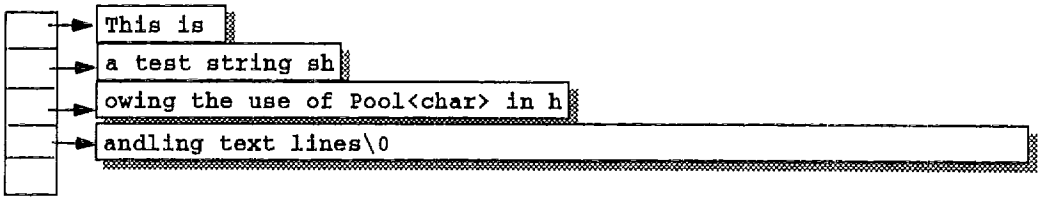

Figure 1. A String in Pool representation.

Figure 1 shows an example use of string by Pool<char>. The size of first segment is *8*. The number of segments currently used is *4*. The length of string is *75*. Totally there are *120* bytes allocated, $120=(2^4-1)*8$.

We give an implementation of pool in C++ template class. The `Pool` template class contains three operations: `grow`, `operator[]`, and `forEachItem`. Operation `grow` returns a reference to a newly allocated space for an element. The index `operator[]` returns the reference of *n-th* element. Operation `forEachItem` is an iterator that accepts a function pointer (`f`) and a pointer of arguments (`arg`). It passes the element and the argument to `f`, "`f(*obj, arg)`".

```
template<class objT>
class Pool
{
public:
  Pool(int log2_init_sz=4, int nindex=16);
  objT& grow();
  objT& operator[] (int n) const;
  void forEachItem(void(*f)(objT&, void*), void* arg=0);
public:
  int log2_segment_size;  // log2(size of first segment)
  int nsegments;          // number of segments
  int nitems;             // number of items
  objT* item;             // next available item in current segment
  objT* limit;            // address after last item
  objT** index;           // support up to 2^nindex - 1
};

template<class objT>
Pool<objT>::Pool(int log2_init_sz=4, int nindex=16) :
      log2_segment_size(log2_init_sz), nsegments(1), nitems(0)
{
  index = (objT**) malloc(nindex*sizeof(void*));
  /* allocate 1st segment */
  item = index[0] = (objT*) malloc( (1<<log2_init_sz)*sizeof(objT));
  limit = item + (1<<log2_init_sz);
}
```

```
template<class objT>
objT& Pool<objT>::grow()
{
  if (item < limit)
    return nitems++, *item++;
  int sz=1<<(log2_segment_size+nsegments); /* allocate next segment */
  item = index[nsegments] = (objT*) malloc( sz*sizeof(objT) );
  limit = item + sz;
  nsegments++;
  return nitems++, *item++;
}

template<class objT>
objT& Pool<objT>::operator[](int n) const
{
  int segment_len = 1<<log2_segment_size;
  int offset = n;
  int indx=0;
  while (offset >= segment_len) {
    offset = offset - segment_len;
    segment_len = segment_len << 1;
    indx++;
  }
  return *(index[indx] + offset);
}

template<class objT>
void Pool<objT>::forEachItem(void (*f)(objT&, void*), void* arg=0)
{
  int segment_len = 1<<log2_segment_size;
  for(int i=0; i<nsegments-1; i++, segment_len=segment_len<<1)
    for(int j=0; j<segment_len; j++)
      f(*(index[i]+j), arg);
  for(objT *obj=index[i]; obj<item; obj++)   // last segment
    f(*obj, arg);
}
```

3. DISCUSSION

Figure 2 summarizes the time complexities of operations on pool, list, and tabular (array) representations. Pool and array can support two kinds of iterators: obverse and reverse. The iterators provided in list representation depend on whether the linked structures are double- or single- linked list. Although all these iterator operations have the same time complexity, the speed is as follows: array > pool > list. Pool is faster than list since the internal loop in iterator of pool is an iteration on a segment, which is in tabular representation.

	list	array	pool
iterators	$O(1)$	$O(1)$	$O(1)$
grow	$O(1)$	--	$O(1)$
index	$O(n)$	$O(1)$	$O(\log n)$

Figure 2. Summary of operations in list, tabular, and pool representations.

All memory chunks requested by pool are in 2^i. The space wasted is *1/2* in worst case and *1/4* in average. This space overhead is the same as buddy system in dynamic memory management [1]. This implies that allocating a collection object in pool has no space overhead if buddy system is used.

4. Conclusion and Applications

We have presented a hybrid data structure called pool for implementations of collection objects. Pool has many potential applications. Here we present a couple of examples in compiler construction:

Pool<char>. In a compiler, the size of a string literal scanned does not have any limitation. A compiler need maintain a data structure flexible for various size of literals. GNU CC [2] uses `obstack` for such literals. When the space allocated for the literal is overflow, it allocates a bigger contiguous space. The contents is then copied into the new space. Pool<char> can provide unbounded space for such literals without moving data. Each character scanned is put in the location returned by the `grow` operation. A `grow` operation may allocate a new segment but does not copy data. Most operations on a literal, e.g., to print a literal, can be done by using `iterator` operation on each character.

Pool<Entry>. The entries in a hash symbol table sometimes need be organized in its insertion order. Consider a symbol table for the definition of a structure. The insertion order of entries in the table is related with the storage layout of a structure. Pool<Entry> provides unbounded space for the unlimited number of entries in a scope. It also maintains a linear (insertion) order on the created entries by their physical adjacency. We have used this data structure in development of a systematic catalog for symbol processing [4].

References

1. Knuth, D. E., *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, 1973.
2. Stallman, R.M., *Using and Porting GNU CC*, version 2.4, Free Software Foundation, June 1992.
3. Uhl, J., and Schmid. H.A., *A Systematic Catalogue of Reusable Abstract Data Types*, LNCS No. 460, Springer-Verlag, 1990, (Ch. 6.2 Tabular Collections), p.117.
4. Wu, P.-C., Lin, J.-H., and Wang, F.-J, "Designing a Reusable Symbl Table Library," Technical report No. CSIE-93-1010, Department of Computer Science and Information Engineering, National Chiao-Tung University, Taiwan, R.O.C., 1993. (Available via ftp.csie.nctu.edu.tw in directory "/papers/tech-report/1993")