Classes currently available in alib

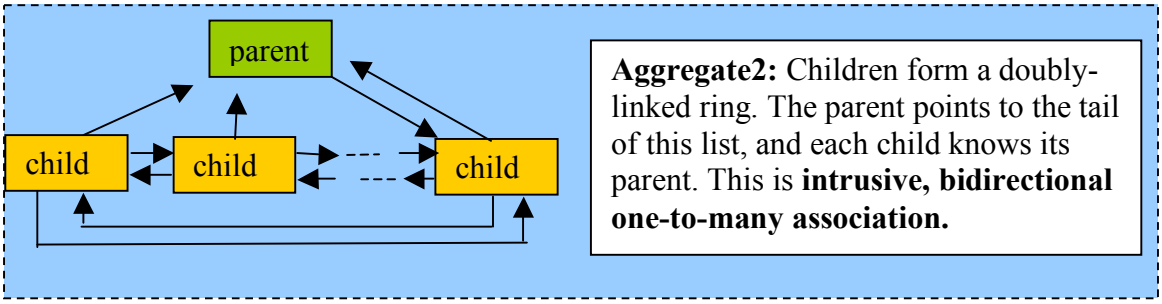
All the sets can be traversed in the same style:

```
    Parent *p; Child *c; Iterator it;
    Forward traversal: for(c=it.fromHead(p); c; c=it.next()){...}
    Reverse traversal: for(c=it.fromTail(p); c; c=it.next()){...}
```

or using a convenient macro ITERATE:

```
    Parent *p; Child *c; Iterator it;
    Forward traversal: it.start(p); ITERATE(it,c){...}
    Reverse traversal: it.start(p); RETRACE(it,c){...}
```

The reverse traversal is available only for doubly-linked sets.



Aggregate2: Children form a doubly-linked ring. The parent points to the tail of this list, and each child knows its parent. This is **intrusive, bidirectional one-to-many association.**

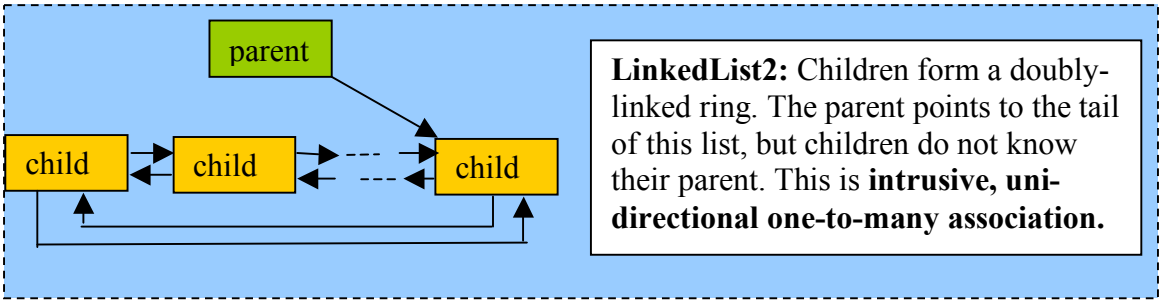
Aggregate2 {

```
public:
    static void addHead(Parent *p, Child *c);
    static void addTail(Parent *p, Child *c);
    static void append(Child *c1, Child *c2); // append c2 after c1
    static void insert(Child *c1, Child *c2); // insert c2 before c1
    static void remove(Child *c); // disconnects c but does not destroy it
    static Parent* const parent(Child *c);
    static Child* const tail(Parent *p);
    static Child* const head(Parent *p);
    static Child* const next(Child *c); // when c it tail, returns NULL
    static Child* const prev(Child *c); // when c it head, returns NULL
    static void sort(ZZsortFun cmpFun, Parent *p);     // see note 1
    static void merge(Child *s,Child *t,Parent *p);    // see note 2
    static void setTail(Parent* p,Child* c,int check); // see note 3
};
```

```
class Aggregate2Iterator {
public:
    Child* fromHead(Parent *p);
    Child* fromTail(Parent *p);
    Child* next();
```

```
    void start(Parent *p);
};
```

NOTES:
(1) cmpFun() compares two children, using the same syntax
    as the similar function required for qsort().
(2) merge() can either merge two rings, or to split one:
    If s and t have different parents, p must be the parent of either s or t,
    and the two rings merge under p. The other parent is left without children.
    If s and t have the same parent, p must be a parent without children, and
    the original ring splits at s and t, with the ring containing t under p.
(3) setTail() resets the tail within the same ring. For check=1, the method
    performs a consistency check which in this case is relatively expensive.
    For check=0, the operation is very fast, but potentially dangerous.



LinkedList2: Children form a doubly-linked ring. The parent points to the tail of this list, but children do not know their parent. This is **intrusive, uni-directional one-to-many association.**

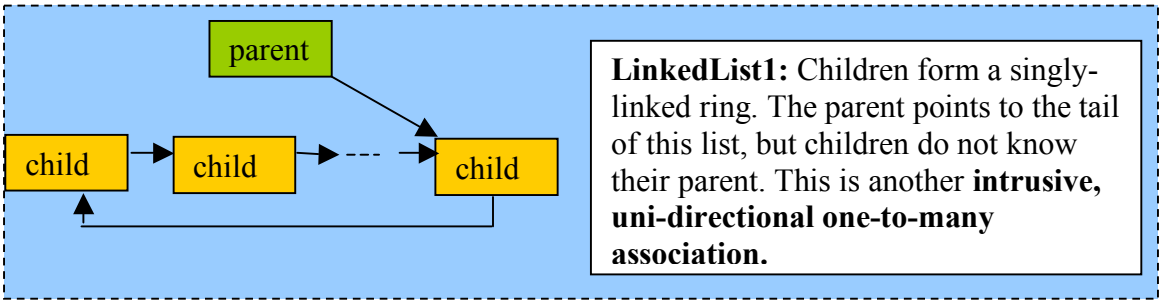
LinkedList2 {

```
public:
    static void addHead(Parent *p, Child *c);
    static void addTail(Parent *p, Child *c);
    static void append(Parent *p,Child *c1, Child *c2); // append c2 after c1
    static void insert(Parent *p,Child *c1, Child *c2); // insert c2 before c1
    static void remove(Parent *p,Child *c); // disconnects c, does not destroy it
    static Child* const tail(Parent *p);
    static Child* const head(Parent *p);
    static Child* const next(Parent *p,Child *c); // when c it tail, returns NULL
    static Child* const prev(Parent *p,Child *c); // when c it head, returns NULL
    static void sort(ZZsortFun cmpFun, Parent *p);     // see note 1
    static void merge(Child *s,Child *t,Parent *p);    // see note 2
    static void setTail(Parent* p,Child* c,int check); // see note 3
};
```

class LinkedList2Iterator {

```
public:
    Child* fromHead(Parent *p);
    Child* fromTail(Parent *p);
    Child* next();
    void start(Parent *p);
};
```

```
NOTES:
(1) cmpFun() compares two children, using the same syntax
    as the similar function required for qsort().
(2) merge() can either merge two rings, or to split one:
    If s and t have different parents, p must be the parent of either s or t,
    and the two rings merge under p. The other parent is left without children.
    If s and t have the same parent, p must be a parent without children, and
    the original ring splits at s and t, with the ring containing t under p.
(3) setTail() resets the tail within the same ring. For check=1, the method
    performs a consistency check which in this case is relatively expensive.
    For check=0, the operation is very fast, but potentially dangerous.
```

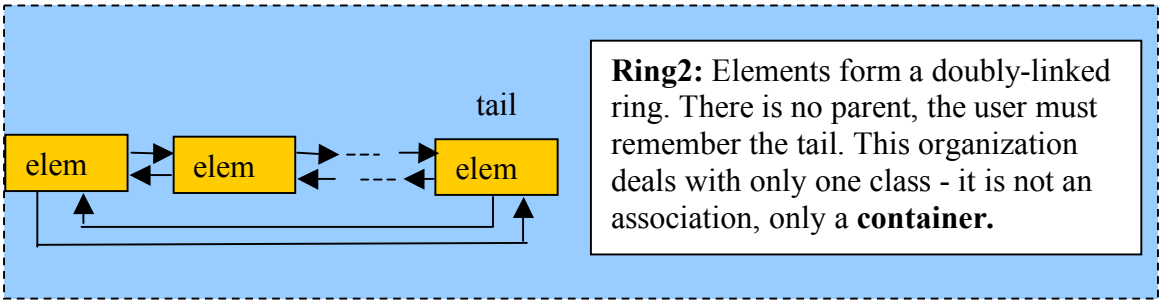


LinkedList1: Children form a singly-linked ring. The parent points to the tail of this list, but children do not know their parent. This is another **intrusive, uni-directional one-to-many association.**

```
LinkedList1 {
public:
    static void addHead(Parent *p, Child *c);
    static void addTail(Parent *p, Child *c);
    static void append(Parent *p,Child *c1, Child *c2); // append c2 after c1
    static void remove(Parent *p,Child *c); // disconnects c, does not destroy it
    static Child* const tail(Parent *p);
    static Child* const head(Parent *p);
    static Child* const next(Parent *p,Child *c); // when c it tail, returns NULL
    static void sort(ZZsortFun cmpFun, Parent *p);     // see note 1
    static void merge(Child *s,Child *t,Parent *p);    // see note 2
    static void setTail(Parent* p,Child* c,int check); // see note 3
};

class linkedList1Iterator {
public:
    Child* fromHead(Parent *p);
    Child* next();
    void start(Parent *p);
};
```

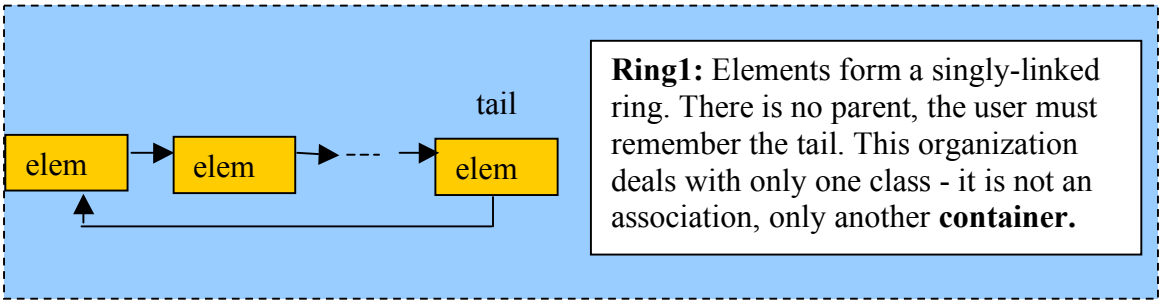

```
NOTES:
(1) cmpFun() compares two children, using the same syntax
    as the similar function required for qsort().
(2) merge() can either merge two rings, or to split one:
```

```
    If s and t have different parents, p must be the parent of either s or t,
    and the two rings merge under p. The other parent is left without children.
    If s and t have the same parent, p must be a parent without children, and
    the original ring splits at s and t, with the ring containing t under p.
(3) setTail() resets the tail within the same ring. For check=1, the method
    performs a consistency check which in this case is relatively expensive.
    For check=0, the operation is very fast, but potentially dangerous.
```



Ring2: Elements form a doubly-linked ring. There is no parent, the user must remember the tail. This organization deals with only one class - it is not an association, only a **container.**

```
Ring2 {
public:
    // unless otherwise stated, methods return the new tail
    static Elem* addHead(Elem *tail, Elem *e);
    static Elem*  addTail(Elem *tail, Elem *e);
    static Elem*  append(Elem *tail,Elem *c1, Elem *c2); // append c2 after c1
    static Elem*  insert(Elem *tail,Elem *c1, Elem *c2); // insert c2 before c1
    static Elem*  remove(Elem *tail,Elem *c); // disconnects c without destroy it
    static Elem* const next(Elem *tail,Elem *c); // when c it tail, returns NULL
    static Elem* const prev(Elem *tail,Elem *c); // when c it head, returns NULL
    static Elem*  sort(ZZsortFun cmpFun, Elem *tail);     // see note 1
    static Elem*  merge(Elem *s,Elem *t,Elem *tail);    // see note 2
};

class Ring2Iterator {
public:
    Elem* fromHead(Elem *tail);
    Elem* fromTail(Elem *tail);
    Elem* next();
    void start(Elem *tail);
};

NOTES:
(1) cmpFun() compares two children, using the same syntax
    as the similar function required for qsort().
(2) merge() can either merge two rings, or to split one:
    If s and t have different parents, p must be the parent of either s or t,
    and the two rings merge under p. The other parent is left without children.
    If s and t have the same parent, p must be a parent without children, and
    the original ring splits at s and t, with the ring containing t under p.
```



Ring1: Elements form a singly-linked ring. There is no parent, the user must remember the tail. This organization deals with only one class - it is not an association, only another **container.**

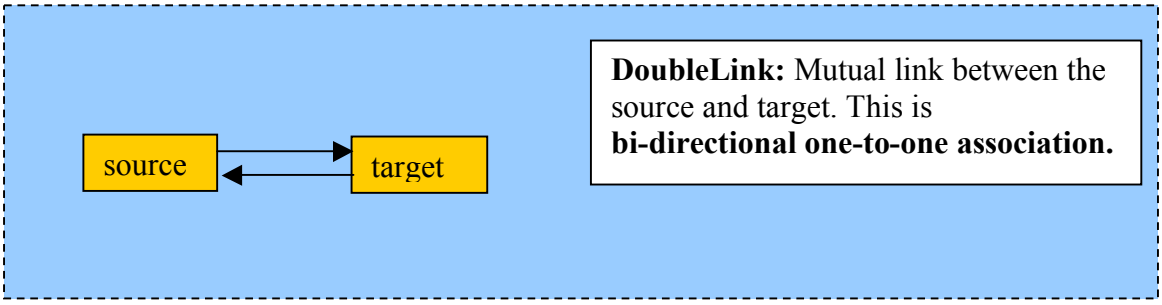
Ring1 {

```
public:
    // unless otherwise stated, methods return the new tail
    static Elem* addHead(Elem *tail, Elem *e);
    static Elem*  addTail(Elem *tail, Elem *e);
    static Elem*  append(Elem *tail,Elem *c1, Elem *c2); // append c2 after c1
    static Elem*  remove(Elem *tail,Elem *c); // disconnects c without destroy it
    static Elem* const next(Elem *tail,Elem *c); // when c it tail, returns NULL
    static Elem*  sort(ZZsortFun cmpFun, Elem *tail);     // see note 1
    static Elem*  merge(Elem *s,Elem *t,Elem *tail);    // see note 2
};
```

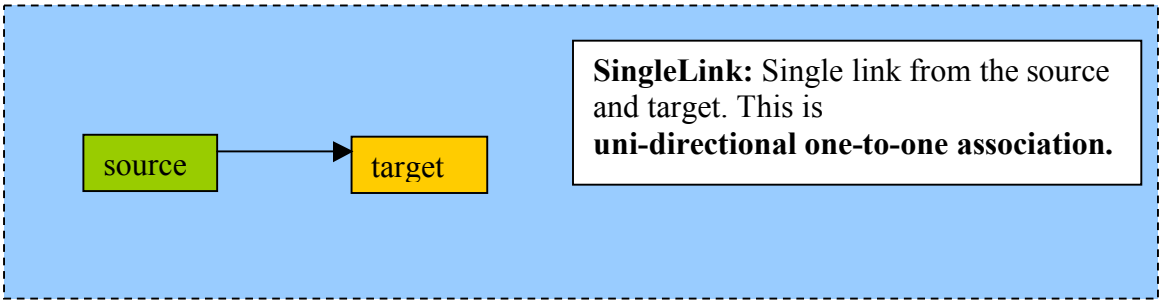
class Ring1Iterator {

```
public:
    Elem* fromHead(Elem *tail);
    Elem* next();
    void start(Elem *tail);
};
```

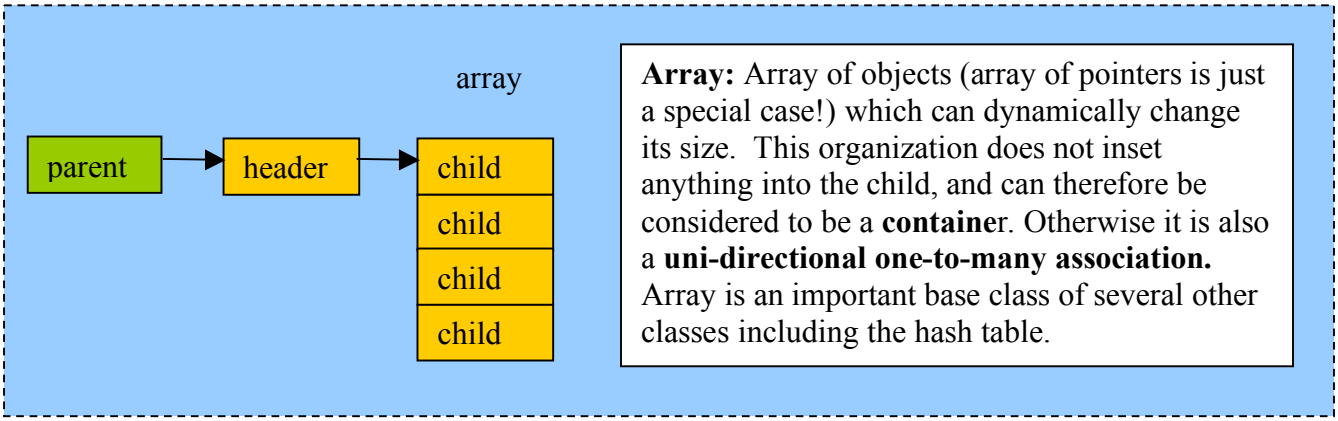
```
NOTES:
(1) cmpFun() compares two children, using the same syntax
    as the similar function required for qsort().
(2) merge() can either merge two rings, or to split one:
    If s and t have different parents, p must be the parent of either s or t,
    and the two rings merge under p. The other parent is left without children.
    If s and t have the same parent, p must be a parent without children, and
    the original ring splits at s and t, with the ring containing t under p.
```




```
class DoubleLink {
public:
    static void add(Source *s,Target *ct;
    static void remove(Source *s);
    static Target* const next(Source *s);
    static Source* const prev(Target *t);
};

// This class does not have any iterator
```




```
class SingleLink {
public:
    static void add(Source *s,Target *ct;
    static void remove(Source *s);
    static Target* const next(Source *s);
};

// This class does not have any iterator
```



Array: Array of objects (array of pointers is just a special case!) which can dynamically change its size. This organization does not inset anything into the child, and can therefore be considered to be a **containe**r. Otherwise it is also a **uni-directional one-to-many association.** Array is an important base class of several other classes including the hash table.

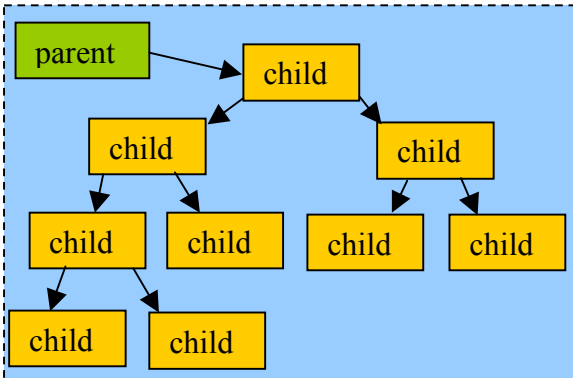
```
class Array {
public:
    static void setMaxSize(unsigned int maxSize){maxSz=maxSize;} // see note 1
    static Child* form(Parent *p,unsigned int const sz,int const incr); //note 2
    static int formed(Parent *p); // 1=array has been formed, 0=not formed
    static void free(Parent *p); // deallocate the header and the array
    static unsigned int capacity(Parent *p); // currently allocated size
    static unsigned int size(Parent *p); // actually used size
    static int increment(Parent *p); // currently used increment
    static Child* get(Parent *p,const unsigned int k);  // a=array[k]
    static void set(Parent *p,const unsigned int k,Child a); // array[k]=a
    static void extract(Parent *p,const int k);          // see note 3
    static void remove(Parent *p,const unsigned int k); // see note 3
    static void insert(Parent *p,const int k,Child *t); // see note 4
    static int reduce(Parent *p); // reduce the allocated size to the used size
    static int reduce(Parent *p,const unsigned int newSz); // see note 5
    static int grow(Parent *p,const unsigned int newSz); // reallocate to newSz
    static void sort(Parent *p,cmpType cmp);   // sort the array using qsort
    static Child* ind(Parent *p,int i); // return pointer to array[i]
    static Child* head(Parent *p); // fast way of getting array[0]
    static void reset(Parent *hp,int newSz,int incr); // reset array parameters
    static void ins(Parent *p,int k,Child* t){insert(hp,k,t);}

    // the following methods keep array ordered, binary search is used
    // in most operations. cmpF() compares two children and has the same
    // syntax as the compare function for sorting (e.g. qsort()).
    static void addOrd(cmpType cmpF,Parent *p,Child *op);
    static void delOrd(cmpType cmpF,Parent *,int k){ extract(hp,k); }
    static void delOrd(cmpType cmpF,Parent *hp,Child *obj);
    static int getOrd(cmpType cmpF,Parent *p,Child *op,int *found); // note 6
};
```

There is no iterator, you traverse the array using integer index:

```
int i,sz; Parent *p; Child *c;
sz=array::size(p);
for(i=0; i<sz; i++){
    c=get(p,i);
    ...
}
```

NOTES:

(1) setMaxSize() sets the size above which no array is allowed to grow.
(2) form() forms the array with the initial size sz. Value of incr controls how the array will grow: for incr>0 sz=sz+incr, for incr<0 sz=sz*(-incr), for incr=0 the array has a fixed size.
(3) extract() removes entry k and shrinks the array without changing the order, while remove() removes entry k and replaces it by they highest index entry. That is fast but changes the order of the array.
(4) Inserts the given object as entry k, and shifts the remaining part of the array.
(5) There are two forms of reduce(). This one forces reallocates the array to the given size, and throws away anything beyond it.
(6) getOrd() makes binary search for op, and returns its index. If there is no match, it returns the index of the object after which it would have to be inserted in order to keep the order (-1 if smaller than the first entry). It returns found=1 when the object is found, found=0 when not.

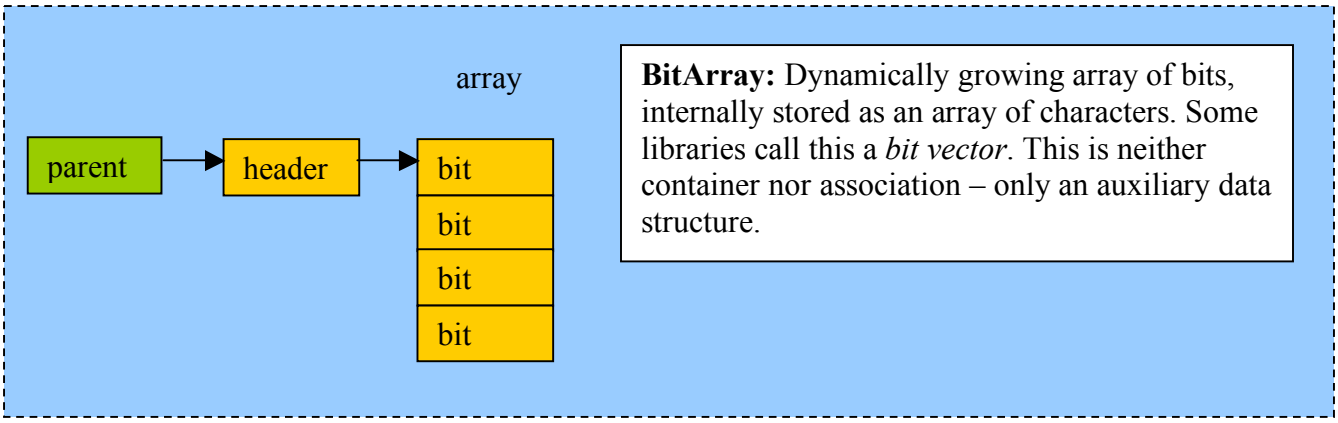


BinaryHeap: Partially sorted binary tree, which allows a fast retrieval of the child with smallest/largest value. This is a special, **partially sorted, uni-directional one-to-many association.** Even though the picture shows a tree, it is internally implemented as an array with smart indexing. The interface reflects this – for example, the heap must be formed before it is used, and its allocated size grows line an array depending on the given increment.

```
When using BinaryHeap, you have to provide two functions:
(1) int (*cmpType)(const void *, const void *);
    compares two children in the same way as the function needed for qsort
(2) void (*bck)(void *, int);
    is an optional function which can report the move of objects within
    the heap, e.g. their array index.

The internal implementation is based on this trick: If we number children row-by-
row 0,1,2,3,... then for entry i, i/2 is the index of its parent and
(2*i+1),(2*i+2) are indexes of its children.
```

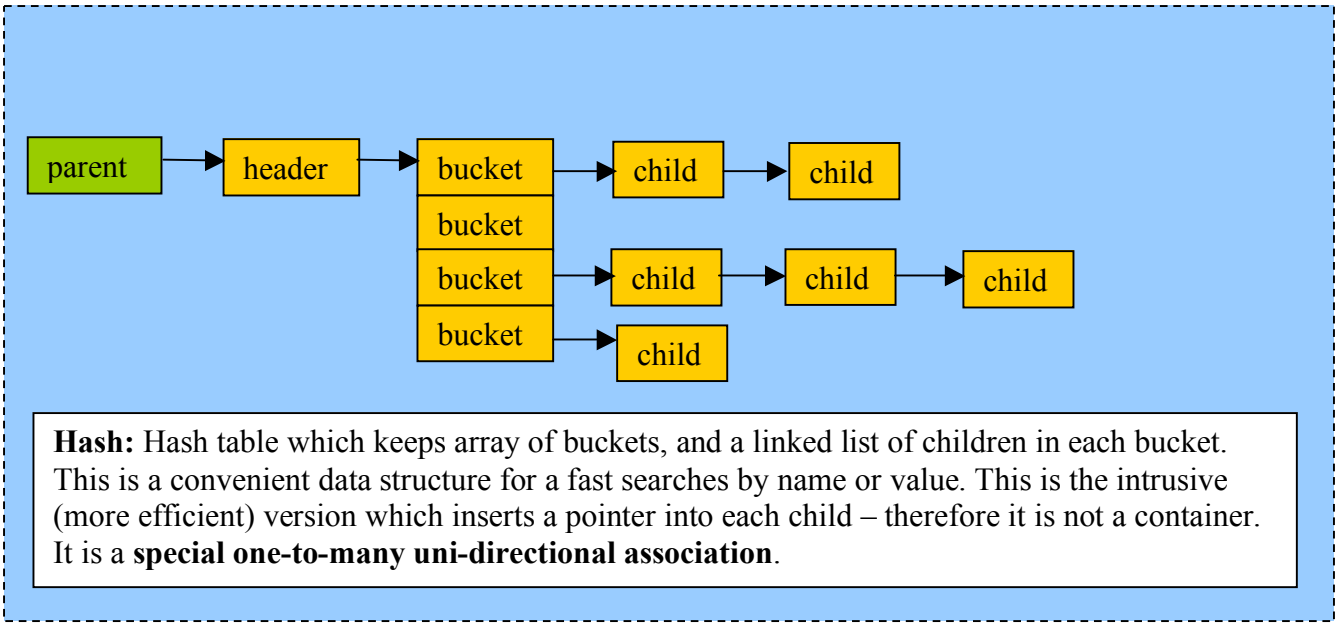
class BinaryHeap {

```
public:
    // Allocate heap for up to sz objects, using increment as for arrays
    static Child* form(Parent *p,unsigned int const sz,int const incr);

    // Free (deallocate) the entire heap
    static void free(Parent *p);

    static int size(Parent *p); // current number of objects in the heap
    static int capacity(Parent *p); // currently allocated space for the heap

    // Insert e into the heap
    static void inHeap(cmpType cmpF,Parent *p,Child *e,bck callback);

    // Copy the top object into e and remove it from the heap.
    // Return 0 when the heap is empty.
    static int outHeap(cmpType cmpF,Parent *p,Child *e,bck callback);

    // Re-sort the heap, assuming that the value for index n has changed.
    static void updHeap(cmpType cmpF,Parent *p,int n,bck callback);

    // Delete entry with index n, and re-sort the heap.
    static void delHeap(cmpType cmpF,Parent *p,int n,bck callback);
};
```



BitArray: Dynamically growing array of bits, internally stored as an array of characters. Some libraries call this a *bit vector*. This is neither container nor association – only an auxiliary data structure.

```
class BitArray {
public:
     // the array must be formed like normal arrays, but all indexing is for bits
    static int formed(Parent *p); // returns 0 when not formed

    static void setMaxSize(unsigned int maxSize); // set size limit
    static int formed(Parent *p); // returns 0 when not formed
    static void free(Parent *p); // free (deallocate) the array
    static unsigned int size(Parent *p); // used number of bits
    static unsigned int capacity(Parent *p); // allocated number of bits
    static int increment(Parent *p); // reset the increment

    static int getBit(Parent *p,int i);
    static int setBit(Parent *p,int i,int val); // set the bit to val
};
```



Hash: Hash table which keeps array of buckets, and a linked list of children in each bucket. This is a convenient data structure for a fast searches by name or value. This is the intrusive (more efficient) version which inserts a pointer into each child – therefore it is not a container. It is a **special one-to-many uni-directional association**.

When you use a hash table, you have to provide two functions (for examples, see the tutorial or the alib\test directory):

```
    static int Hash::cmp(Child *pc,Child *c2);
```

which returns 0 if the objects have the same key, and

```
    static int Hash::hash(Child *c,int hashSz);
```

which converts key to a bucket number, assuming that hashSz is the number of buckets. If you do not have your own favourite algorithm for this, use default function hashString() or hashInt() already provided. For example:

```
    class Child {
        int key;
    public:
        ZZ_Child ZZds;
        int getKey(){return key;}
    };

    static int Hash::hash(Child *c,int hashSz){
        return hashInt(c->getKey(),hashSz);
    }
```

Note that this class has an iterator which allows to traverse either a selected bucket or the entire hash table:

```
    Child *d; Parent *p;
    HashIterator it;
    for(c=it.first(p,i); c; c=it.next()){...}
    // where i is the bucket index, -1 will traverse the entire hash table
```

Since, internally, this class is derived from Array, its interface has a similar style. The hash table must be first formed in order to be used. However, the table does not re-allocate itself, and it is left to the user to monitor its

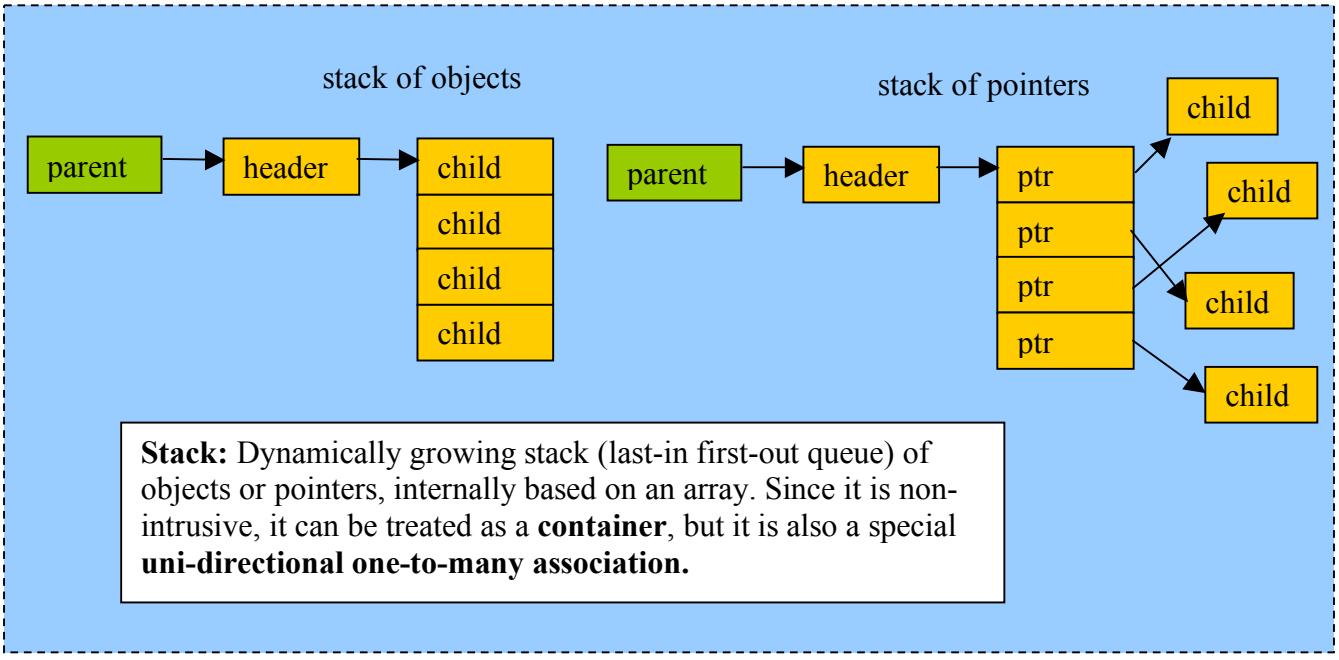
loading and increase its size when needed. This usually does not take much code and results in a more intelligent and more efficient software.

class Hash {

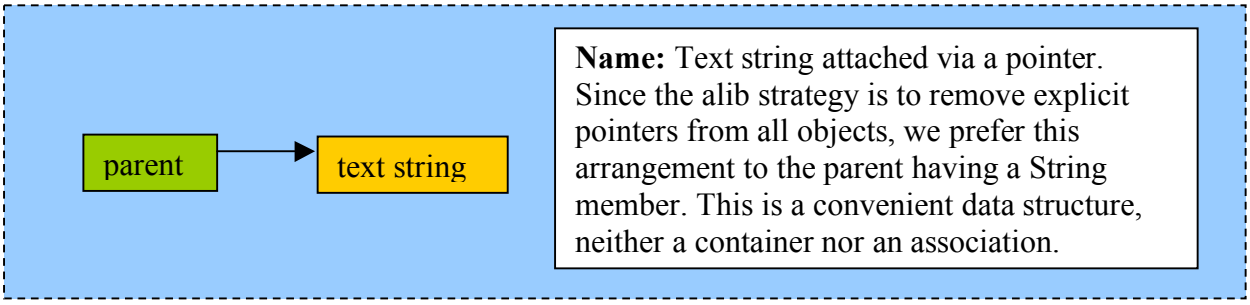
```
public:
    static void** form(Parent *p,int const sz); // sz=number of buckets
    static int formed(Parent *p); // returns 0 when table not formed
    static int size(Parent *p, int *popCount);   // see note 1
    static void free(Parent *p); // free (deallocate) the entire hash table
    static int resize(Parent *p, int newSz); // resize to newSz buckets, reload
    static Child* get(Parent *p, Child *obj);    // see note 2
    static int add(Parent *p, Child *obj); // load obj into the hash table
    static Child* remove(Parent *p, Child *obj); // see note 3

    // convenient default functions to use when coding hash()
    // ------------------------------------------------
    static int hashString(char *s,int hashSz);
    static int hashInt(int val,int hashSz);
};

class HashIterator {
public:
    Child* first(Parent *p,int i);  // traverse bucket i, i<0 for entire table
    Child* const next();

};
```

NOTES:

(1) size returns the number of bucket, but also the total number of objects in the table (popCount).
(2) get() searches the table and returns the object with the same key as the auxilliary object obj. It returns NULL if such object not found.
(3) remove() removes the object with the key identical to the auxiliary object obj, or obj itself.

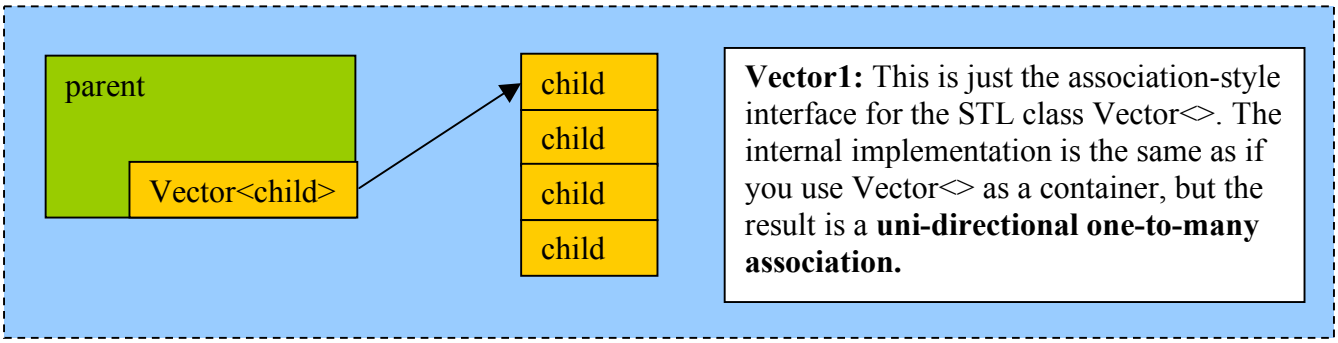



```
class Stack {
public:
    static Child* form(Parent *p,unsigned int const sz,int const incr); // note 1
    static void push(Parent *p,Child* c);
    static Child* pop(Parent *p);
    static void free(Parent *p); // free and deallocate the entire stack
    static int size(Parent *p,int* wMark,int* incr);                // see note 2
};

NOTES:

(1) Form the stack as you would form an array: sz=initial size,
    with the dynamic grow: for incr>0 sz=sz+incr, for incr<0 sz=sz*(-incr), for
    incr=0 the size is fixed.
(2) size() returns allocated size, but also current number of entries (wMark) and
    currently used increment.
```



Name: Text string attached via a pointer. Since the alib strategy is to remove explicit pointers from all objects, we prefer this arrangement to the parent having a String member. This is a convenient data structure, neither a container nor an association.

```
class Name {
public:
    static void add(Parent *p, char *c);
    static char* get(Parent *p);
    static char* remove(Parent *p);
    static int compare(Parent *p1, Parent *p2);
};
```



Vector1: This is just the association-style interface for the STL class Vector<>. The internal implementation is the same as if you use Vector<> as a container, but the result is a **uni-directional one-to-many association.**

Classes Vector1 and Vector2 are an example how you can reuse container classes from a common library such as STL while treating them as or expanding them to associations. Except for the interface, the STL class Vector<> is similar to our class Array, and is typically used as a member on the parent object:

```
    class Department {
        Vector<Employee> empl;
        ...
    }

    Department *d; Employee *e;
    e=d->empl.at(7); // get entry at position 7
```

Note that when you read the last line, it is: *Go to d, get its vector and call its function at() with 7.*

The alib class Vector1 does the same thing, but hides the STL Vector<> and provides a fully-typed association-style interface:

```
class Vector1 {
public:
    typedef vector<Child>::iterator iterator;
```

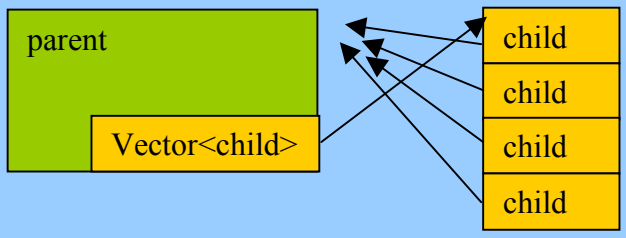
```
    static iterator begin(Parent *p);
    static iterator end(Parent *p);

    Child *at(Parent *p, Child *c); // replaces Vector<Child>::at(Child *c)
    // .. similar change for all other methods of the STL Vector<>

};
```

Example of use:

```
    class Department {
    public:
        ZZ_Department ZZds;
        ...
    };
    class Employee {
    public:
        ZZ_Employee ZZds;
        ...
    };

    Association Vector1<Department,Employee> empl;

    Department *d; Employee *e;
    e=empl::at(d,7);
```

When you read the last line, the message is a bit different and I believe more meaningful: *Go to the vector which is on d and get the element index 7.* Compare this with the statement we got when using Vector<> as member above.

Note that the current Vector1 class in alib is only a skeleton which shows the concept. Someone will have to spend time going through some 40+ methods of Vector<> plus numerous operators to test the result. Some programmers think that data structures with a multitude of methods are great. I hate them. Unless you memorize all the methods of a class you cannot properly use it, and what is even worth – you cannot maintain software in which the class was used by someone else.



Vector2: This is a class which is not a container and therefore isn't included in STL. It uses the STL class Vector<> to keep an array of children who all know the their parent. The result is a **bi-directional one-to-many association.**

```
class Vector2 {
public:
    typedef vector<Child>::iterator iterator;
    static iterator begin(Parent *p);
    static iterator end(Parent *p);

    static Parent *getParent(Child *c); // the only new function
    static Child *at(Parent *p, Child *c);
    // .. all other methods of STL vector
};
```

NOTE: Internally, Vector2 is derived from Vector1, with most of the methods remaining without any change. Unfortunately, that means it is also just a skeleton until all the methods of Vector1 are filled in.