Appendix A
A Summary of the PAD Library

Tables A.1–A.19 of this appendix summarize the data structures and most important functions of the current version of PAD. Although this version is not compatible with earlier versions of PAD, the implementations are essentially the same. The user of the library is encouraged to regularly check the Web page for the book, where an updated summary of the library is maintained. New contributions to PAD can be reported via the Web page. See Appendix **??** for details.

The PAD datatype definitions, macro definitions, and prototype declarations are contained in the PAD header file `pad.h`, which must be included by any program using functionality from PAD. The actual code of the library is contained in the file `pad.c`, which can be separately compiled, and subsequently linked with the application program.

Table A.1 General Purpose PAD Macros

	Special Variables
`PADp`	Shared variable holding the processor count; used by the parallel iterators
`DECp(p)`	Declare and set shared processor count `PADp`
`SETp(p)`	Set shared processor count `PADp`
	Parallel Iterators
`par_array(i,n)`	Parallel array iterator; for processor `$`, the private variable `i` attains the values $\$, \$ + \mathrm{PADp}, \$ + 2\mathrm{PADp}, \ldots$; the processor count `PADp` must have been set
`rap_array(i,n)`	Parallel array iterator, reverse direction; for processor `$`, the private variable `i` attains the values $n - 1 - \$, n - 1 - \$ - \mathrm{PADp}, n - 1 - \$ - 2\mathrm{PADp}, \ldots$; the processor count `PADp` must have been set
`par_item(e,L)`	List iterator
`par_nodes(v,T)`	Node iterator for trees and graphs
`par_edges(e,T)`	Edge iterator for trees and graphs

Table A.2 PAD Datatypes

Type Casting Prototypes for Functions	
`UP2`	Type of update function with two parameters
`UP3`	Type of update function with three parameters
`CM2`	Type of comparison function with two arguments
`RT1`	Type of function returning `void` pointer
PAD Datatypes	
`bool`	Boolean type, values `TRUE` and `FALSE`
`Array`	Generic array; start address in field `sa`, number of elements in `nn`, element size in `es`
`Range`	Range query data structure
`ListItem`	List item with successor and predecessor pointers `pred` and `succ`, rank in field `rank`; pointer to contents in `content`
`List`	List; length of list in field `nn`
`TreeNode`	Tree node with pointers to the segment of adjacent nodes in the `TreeEdge` array in `firstedge` and `lastedge`, parent pointer in `parent`; traversal numbers `level`, `post`, `pre`, `desc`, left- and rightmost occurrence of node on Euler tour in `l` and `r`; pointer to contents of node in `content`
`TreeEdge`	Pointer to tail and head node in fields `tail` and `head`, reverse edge pointer in `rev`, pointer to next edge on Euler tour in `eulertour`; pointer to contents of edge in `content`
`Tree`	Number of nodes in field `nn`, number of edges in `mm`. Pointers to node and edge arrays in fields `nodes` and `edges`, root node in `root`; array of list items on Euler tour in `euler`, range structure for lowest common ancestor querying in `lcaquery`
`GraphNode`	Graph node with pointers to the segment of adjacent nodes in the `GraphEdge` array in `firstedge` and `lastedge`, parent pointer in `parent`; connected component index in field `cc`. Pointer to contents of node in `content`
`GraphEdge`	Pointer to tail and head node in fields `tail` and `head`, reverse edge pointer in `rev`; index of biconnected component in field `bc`; pointer to contents of edge in `content`
`Graph`	Number of nodes in field `nn`, number of edges in `mm`. Pointers to node and edge arrays in fields `nodes` and `edges`
`DictItem`	Dictionary item; key in field `key`, number of children in `no_of_children`, number of leaves in `no_of_leaves`, pointer parent in dictionary in `father`; pointer to contents of dictionary leaf in `children[0]`
`keytype`	Type of key in dictionary

Table A.3 Elementary Operations on Generic Arrays

`ARR(a,i)`	Access macro for generic arrays; returns `(char *)` pointer to `i`th element of array `a`

```
sync void NewArray(sh Array *x, sh int n, sh int size);
```

`NewArray` allocates space for `n` elements of size `size` on the permanent shared heap; array `x` must have been declared prior to the call

```
sync void DelArray(sh Array *x)
```

`DelArray` frees space on the permanent shared heap allocated for array `x`

```
void ArraySegment(Array *a, int i, int p, Array *b);
```

`ArraySegment` creates `i`th segment of `p` segments for array `a`; the number of elements in the segment will be either ⌊`a->nn/p`⌋ or ⌈`a->nn/p`⌉; an array representing the `i`th segment is returned in `b`; space for the `Array` structure `b` must have been allocated prior to the call

```
void ArraySlice(Array *a, int ix, int nn, Array *b);
```

`ArraySlice` creates a segment of array `a` with `nn` elements, starting at index `ix` (of `a`); space for the `Array` structure `b` must have been allocated prior to the call.

```
sync void ArrayCopy(sh Array *in, sh Array *out);
```

`ArrayCopy` copies all elements of array `in` to array `out`; the `out` array must have been allocated by `NewArray` prior to the call

Table A.4 Prefix-Sums and Related Operations for Generic Arrays

An associative function $\oplus$ on the base type of the argument array must be implemented as an asynchronous three-argument *update function* `fu`, implementing the assignment

$$*\mathtt{x} = \oplus(*\mathtt{y}, *\mathtt{z});$$

when called with pointers to `x`, `y` and `z`

```
sync void Prefix(sh Array *inout, UP3 fu);
```

Inclusive prefix-sums over the sequence of elements stored in array `inout` with respect to update function `fu`; the sequence of prefix-sums is returned in `inout`; this is the implementation of choice

```
sync void RecPrefix(sh Array *inout, UP3 fu);
```

Recursive implementation of the generic, inclusive prefix-sums operation

```
sync void Suffix(sh Array *inout, UP3 fu);
```

Generic, inclusive suffix-sums over the sequence of elements stored in array `inout` with respect to update function `fu`

```
sync sh int SegPrefix(sh Array *inout, UP3 fu,
                      sh int marker[]);
```

Segmented prefix-sums over the sequence of elements stored in array `inout` with respect to update function `fu`; start of a new segment at position `i` is indicated by `marker[i]` equal to 1; `marker[i]` must be 0 otherwise

```
sync int Locate(sh const void *y, sh Array *x, CM2 eq);
```

`Locate` searches in unordered array `x` for an element `y`; equality is defined by the function `eq`, called with pointers to two elements of the base type of array `x`; returns an index `i` such that `y` is equal to `ARR(x,i)`, if `y` is present in `x`, `-1` otherwise

```
sync void Permute_array(sh Array *in, sh Array *out,
                        sh int index[]);
```

The `i`th element of array `in` is moved to position `index[i]` in array `out`; space for array `out` must have been prior to the call by `NewArray`

Table A.5 Search, Merge and Sort Operations for Generic Arrays

Order is defined by a comparison function `cmp`, that must implement a < relation on the base type of the argument arrays; it is called with pointers to elements of this type

```
int Bin_search_low(void *y, Array *x, CM2 cmp);
int Bin_search_high(void *y, Array *x, CM2 cmp);
```

Binary search for `y` in array `x`; the `Bin_search_low` function returns an index $0 \leq$ `i` $\leq$ `n` such that `ARR(x,i-1)` $<$ `y` $\leq$ `ARR(x,i)`; the `Bin_search_high` function returns an index $-1 \leq$ `i` $<$ `n` such that `ARR(x,i)` $\leq$ `y` $<$ `ARR(x,i+1)`

```
sync int Ordered_search_low(void *y, Array *x, CM2 cmp);
sync int Ordered_search_high(void *y, Array *x, CM2 cmp);
```

Parallel search for `y` in array `x`; specification as for binary search

```
void Seq_merge(Array *x, Array *y, Array *z, CM2 cmp);
void Seq_mergesort(Array *x, CM2 cmp);
```

Sequential merge and merge sort for generic arrays

```
sync void SmallMerge(sh Array *x, sh Array *y, sh Array *z,
                     CM2 cmp);
```

`SmallMerge` merges ordered arrays `x` and `y` into array `z`; the number of elements in array `x` should be at most `#` (only the first `#` elements of `x` take part in the merge); space for array `z` must have been allocated prior to the call

```
sync void Merge(sh Array *x, sh Array *y, sh Array *z,
                CM2 cmp);
```

`Merge` merges ordered arrays `x` and `y` into array `z`

```
sync void Mergesort(sh Array *x, CM2 cmp);
```

Parallel merge sort

Table A.6 Inclusive and Exclusive Prefix-Sums Operations for Integer Arrays

Integer arrays are represented by a pointer to the zeroth array element, and the number of elements; these are the basic parameters for all integer operations

```
void seq_add(int in[], const int n, int out[], int offset);
```

Sequential inclusive prefix-addition with offset

```
sync void prefix_add(sh int in[], sh const int n,
                     sh int out[], sh int offset);
sync void prefix_max(sh bool in[], sh const int n,
                     sh bool out[], sh bool max);
sync void prefix_and(sh bool in[], sh const int n,
                     sh bool out[], sh bool term);
sync void prefix_or(sh bool in[], sh const int n,
                    sh bool out[], sh bool term);
```

Parallel inclusive integer prefix-sums operations for addition, maximum, bitwise AND, and bitwise OR

```
sync int preprefix_add(sh int in[], sh const int n,
                       sh int out[], sh int offset);
sync int preprefix_and(sh bool in[], sh const int n,
                       sh bool out[], sh bool term);
sync int preprefix_or(sh bool in[], sh const int n,
                      sh bool out[], sh bool term);
sync int preprefix_max(sh bool in[], sh const int n,
                       sh bool out[], sh bool max);
```

Parallel exclusive integer prefix-sums operations for addition, maximum, bitwise AND, and bitwise OR

```
sync void suffix_add(sh int in[], sh const int n,
                     sh int out[], sh int offset);
sync void suffix_max(sh bool in[], sh const int n,
                     sh bool out[], sh bool max);
sync void suffix_and(sh bool in[], sh const int n,
                     sh bool out[], sh bool term);
sync void suffix_or(sh bool in[], sh const int n,
                    sh bool out[], sh bool term);
```

Parallel exclusive integer suffix-sums operations for addition, maximum, bitwise AND, and bitwise OR

Table A.7 Colored and Segmented Prefix-Sums Operations for Integer Arrays

```
sync void colorprefix_add(sh int x[], sh const int n,
                          sh int color[], sh const int c,
                          sh int ccount[]);
sync void colorprefix_max(sh int x[], sh const int n,
                          sh int color[], sh const int c,
                          sh int ccount[]);
sync void colorprefix_and(sh int x[], sh const int n,
                          sh int color[], sh const int c,
                          sh int ccount[]);
sync void colorprefix_or(sh int x[], sh const int n,
                         sh int color[], sh const int c,
                         sh int ccount[]);
```

Parallel inclusive colored prefix-sums operations for addition, maximum, bitwise AND, and bitwise OR; each element of `x` has a corresponding color `color`, which is an integer in the range from `0` to `c-1`; the sum of the `x` elements with color `k` is returned in `ccount[k]`

```
sync int segprefix_add(sh int x[], sh const int n,
                       sh int y[], sh int marker[]);
sync int segprefix_max(sh int x[], sh const int n,
                       sh int y[], sh int marker[]);
sync int segprefix_and(sh int x[], sh const int n,
                       sh int y[], sh int marker[]);
sync int segprefix_or(sh int x[], sh const int n,
                      sh int y[], sh int marker[]);
```

Parallel inclusive segmented prefix-sums operations for addition, maximum, bitwise AND, and bitwise OR; start of a new segment at index `i` is marked by having `marker[i]` set to `1`, `0` otherwise

Table A.8 Miscellaneous Operations for Integer Arrays

```
void arraysegment(const int n, const int i, const int p,
                  int *ix, int *nn);
```

`arraysegment` segments an integer range from `0` to `n−1` into `p` segments of size either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$; returns in `ix` and `nn` the start and size of the `i`th such segment

```
sync void permute_array(sh int in[], sh int out[],
                        sh const int n, sh int index[]);
```

`permute_array` moves `in[i]` to `out[index[i]]` for all `i` between `0` and `n−1`

```
sync void random_array(sh int x[], sh const int n,
                       sh int seed[], sh const int s);
```

`random_array` generates a random integer array using the pseudo random-number generator of Fork; an array of `s` seeds is provided as input; this makes it possible to reproduce the generation independently of the number of processors; for this `s` should be greater than or equal to the maximum number of processors with which the function will be called

```
sync int locate(sh const int y, sh int x[], sh const int n);
```

Parallel search in an unordered integer array; `locate` returns an index `i` with `y` equal `x[i]` if `y` exists in `x`, `−1` otherwise

Table A.9 Search, Merge, and Sorting Operations for Integer Arrays

```
int bin_search_low(int y, int *x, int n);
int bin_search_high(int y, int *x, int n);
sync int ordered_search_low(sh int y, sh int x[], sh int n);
sync int ordered_search_high(sh int y, sh int x[], sh int n);
```

Sequential and parallel search operations for integer arrays; the specification is as for generic arrays

```
void seq_merge(int x[], const int n, int y[], const int m,
               int z[]);
void seq_mergesort(int in[], const int n);
```

Sequential merge and mergesort

```
sync void smallmerge(sh int *x, sh int n, sh int *y, sh int m,
                     sh int *z);
sync void merge(sh int *x, sh int n, sh int *y, sh int m,
                sh int *z);
```

Parallel merge for integer arrays; function `smallmerge` should be used if `n` is smaller than the number `#` of calling processors

```
sync void mergesort(sh int x[], sh const int n);
```

Parallel merge sort for integer arrays

```
sync void smallperm(sh int keys[], sh const int n,
                    sh const int c, sh int rank[]);
```

`smallperm` computes the rank of an array of integer `keys`, each having value between `0` and `c−1`

```
sync void smallsort(sh int x[], sh int n, sh int c);
```

`smallsort` sorts an array `x` of integers in the range from `0` to `c−1`

Table A.10 Range Query Data Structure

```
sync void NewRange(sh Range *range,
                   sh const int n, sh const int size);
```

`NewRange` allocate space on the permanent shared heap for range query data structure over an array with `n` elements of size `size`; space for the `range` structure itself must have been allocated prior to the call

```
sync void DelRange(sh Range *range);
```

`DelRange` frees space allocated for `range`

```
sync void RangePreproc(sh Array *in, sh Range *range, UP3 fu);
```

Preprocess input array `in` for querying with respect to an associative function implemented by the update function `fu`; a query structure is built up in `range`

```
void RangeQuery(int i, int j, Range *query, void *answer,
                UP3 fu);
```

Query for the value of the associative function implemented by `fu` on the preprocessed input array from index `i` to index `j`

Table A.11 Lists and List Macros

Lists are represented as arrays of `ListItems`, with pointers to successor and predecessor items; a list item has a pointer to the content of the item; the content may be an element of a linked structure, in which case a pointer from element to list item may be available; the access macros give access from list content to information stored in the list item

Access Macros for Lists

`ELT(li,i)`	`((li)[(i)].content)`	Pointer to `i`th element
`RANK(e)`	`((ListItem *)(e))->rank`	Rank of element `e`
`NEXT(e)`	`((ListItem *)(e))->succ->content`	Successor of `e`
`PREV(e)`	`((ListItem *)(e))->pred->content`	Predecessor of `e`
`TAIL(e)`	`(((ListItem *)(e))->succ==NULL)`	`e` is last element
`HEAD(e)`	`(((ListItem *)(e))->pred==NULL)`	`e` is first element

Table A.12 Elementary List Operations

```
sync void NewList(sh List *list, sh int n);
```

`NewList` allocates space on the permanent shared heap for a list of `n` list items

```
sync void DelList(sh List *list);
```

`DelList` frees the space previously allocated for `list`

```
sync void ListMake(sh List *list, RT1 next, UP2 up);
```

`ListMake` makes a list from a linked structure of elements; prior to the call, each list item must have its `content` pointer set to point to a (unique) element of the linked structure; the successor pointer of the linked structure must be accessible through function `next` (value) and `up` (update)

```
sync void ListRestore(sh List *list, UP2 up);
```

`ListRestore` restores linked structure from list

```
sync void ListCopy(sh List *l1, sh List *l2);
```

`ListCopy` copies list `l1` to `l2`; pointers from elements to list items are not changed

```
sync void ListMove(sh List *l1, sh List *l2, UP2 up);
```

`ListMove` copies list `l1` to `l2`; pointers from elements to list items are changed, such that the elements poiting to item `i` of list `l1` prior to the call now points to item `i` of `l2`

```
sync void ListReverse(sh List *list);
```

`ListReverse` reverses `succ` and `pred` pointers of all items of `list`

```
sync int ListEnds(sh List *list,
                  sh ListItem *heads[], sh ListItem *tails[]);
```

`ListEnds` stores items with `NULL` successor in `heads`, and items with `NULL` predecessor in `tails`; space for these arrays of pointers must have been allocated before the call

```
sync void ListSplice(sh List *list, sh ListItem *heads[],
                     sh ListItem *tails[], sh int k);
```

`ListSplice` splices `k` sublists in `list` together; sublist end `head[i]` is spliced to sublist start `tail[i+1]`

Table A.13 List Ranking Operations

`void Seq_rank(List *list)`

`Seq_rank` ranks `list` sequentially; rank of each item is computed in field `rank`

`void Seq_rank_sub(ListItem *active[], int n);`

`Seq_rank_sub` ranks sublist represented by array of `ListItem` pointers sequentially

`sync void ListRank(sh List *list);`

The PAD parallel list ranking operation

`sync void ListArrayOrder(sh List *list, UP2 up);`

`ListArrayOrder` permutes `list` into array order: list item with rank `rank` is moved to index `list->nn-1-rank`

`sync void ListRankOrder(sh List *list, UP2 up);`

`ListRankOrder` permutes `list` into rank order: list item with rank `rank` is moved to index `rank`

`sync void Wyllie(sh List *list)`

`Wyllie` ranks list in parallel by pointer jumping

`sync void Wyllie_sub(sh ListItem *active[], sh int n)`

`Wyllie_sub` ranks sublist in parallel by pointer jumping

`sync void BlockRank(sh List *list)`

List ranking for lists that are long relative to the number of calling processors

Table A.14 List Generators

```
sync void ListPermute(sh int permutation[], sh List *list,
                      UP2 up);
```

`ListPermute` moves `i`th list item to position `permutation[i]`, and update the element pointer accordingly for all items

```
sync void ListForce(sh int cycle[],
                    sh List *list, sh ListItem *head);
```

`ListForce` sets successor and predecessor pointers according to the cyclic permutation `cycle`; the successor of the item at index `cycle[i]` becomes the item at index `cycle[i+1]`

```
sync void SeqListForce(sh List *list);
```

`SeqListForce` forces a *sequential list*; the successor of item `i` becomes item `i+1`

```
sync void MeshlistForce(sh List *list);
```

`MeshlistForce` forces a *mesh list*; the successor of item `i` becomes item `#+i` (modulo `#`)

```
sync void RanListForce(sh List *list, sh ListItem *head);
```

`RanListForce` forces a random list; successor pointers are determined by a random cyclic permutation

Table A.15 Elementary Operations on Trees

Access Macros for Trees and Graphs		
`NODE(i,t)`	`((t)->nodes+(i))`	`i`th node
`NODE_NO(v,t)`	`((v)-(t)->nodes)`	Index of `v`
`EDGE(i,t)`	`((t)->edges+(i))`	`i`th edge
`EDGE_NO(e,t)`	`((e)-(t)->edges)`	Index of `e`
`LEAF(v)`	`((v)->children[0]==NULL)`	`v` is leaf
`DOWN(e)`	`(((e)->head->parent==(e)->tail))`	`e` is down edge
`UP(e)`	`(((e)->tail->parent==(e)->head))`	`e` is up edge
	(The latter three macros for rooted trees only)	

```
sync void NewTree(sh Tree *t, sh const int n);
```

`NewTree` allocates space for a tree `t` with `n` nodes (and `2n-1` edges) on the permanent shared heap

```
sync void DelTree(sh Tree *t);
```

`DelTree` frees space allocated for tree `t`

```
sync void TreeCopy(sh Tree *t1, sh Tree *t2);
```

`TreeCopy` copies tree `t1` to `t2`

```
sync void TreeCompact(sh Tree *t);
```

`TreeCompact` compacts tree `t` by removing nodes and edges that have been marked for deletion; a node is marked for deletion if its `firstedge` pointer is set to `NULL`; an edge is marked for deletion if either its `atil` or `head` is set to `NULL`

```
sync void MarkSubtree(sh Tree *t, sh TreeNode *v);
```

`MarkSubtree` marks all nodes and edges in subtree rooted at `v` for deletion

```
sync void TreeSmallRev(sh Tree *t);
```

`TreeSmallRev` constructs reverse edge pointers for all edges of `t` by table lookup; appropriate if enough space is available, i.e. `t` is small

```
sync void TreeRev(sh Tree *t)
```

`TreeRev` constructs reverse edge pointers for all edges of `t` by sorting

```
TreeEdge *TreeNextEdge(TreeEdge *e, Tree *t);
```

`TreeNextEdge` returns the cyclically next edge after `e` on the adjacency list of the tail node of `e`

Table A.16 Euler Tour Technique and Related Operations on Trees

```
sync void TreeEuler(sh Tree *t);
```

`TreeEuler` constructs an Euler tour on `t` by setting the `eulernext` pointer of each tree edge; precondition: the reverse edge pointers for all edges of `t` have been set correctly

```
sync void TreeRoot(sh Tree *t, sh TreeNode *root);
```

`TreeRoot` roots `t` at node `root`, and orient all edges of `t` towards `root`; after the call the list in array `euler` will be an Euler tour of `t` in array order

```
sync void BinTreeContract(sh Tree *t,
                          UP3 rk, UP3 rkrt, UP3 unrk);
```

Generic tree contraction on rooted, binary tree `t`; rake, unrake and root rake-unrake functions are supplied in the call, each a three-argument function to be called with pointers to a leaf, its sibling, and its parent

```
sync void BinTreeHeight(sh Tree *t);
```

Height computation on rooted, binary tree `t`; the height of each node is computed in the `height` field of `TreeNode`

```
sync void TreeLCApreproc(sh Tree *t);
```

`TreeLCApreproc` preprocesses tree `t` for lowest common ancestor querying; the query structure is stored in the `lcaquery` field of `t`

```
TreeNode *TreeLCA(Tree *t, TreeNode *u, TreeNode *v);
```

`TreeLCA` returns the lowest common ancestor of nodes `u` and `v` of tree `t`

```
sync void *TreeMakebin(sh Tree *s,
                       int dummynode, int dummyedge,
                       sh Tree *t);
```

`TreeMakebin` makes `t` binary and complete by insertion of dummy nodes and edges

```
sync void *CompleteTree(sh Tree *t, sh const int n,
                        sh const int degree,
                        sh const int children);
```

`CompleteTree` generates a skewed, complete tree with `n` nodes; each node attempts to get `children` children, of which `degree` again will be skewed, complete trees

Table A.17 Graph Operations

Similar data structure to that for trees; the same iterators and access macros can be used

```
sync void NewGraph(sh Graph *g,
                   sh const int n, sh const int m);
```

`NewGraph` allocates space on the permanent shared heap for graph `g` with `n` nodes and `m` edges; space for the `Graph` structure `g` must have been allocated prior to the call

```
sync void DelGraph(sh Graph *g);
```

`DelGraph` frees the space allocated for `g`

```
sync void GraphCopy(sh Graph *t1, sh Graph *t2);
```

`GraphCopy` copies graph `g1` to `g2`; space for `g2` must have been allocated by a call to `Newgraph` prior to the call

```
GraphEdge *GraphNextEdge(GraphEdge *e, Graph *t);
```

`GraphNextEdge` returns the cyclically next edge after `e` on the adjacency list of the tail node of `e`

```
sync void GraphCC(sh Graph *g);
```

`GraphCC` determines the simple connected components of graph `g`; two nodes `u` and `v` belong to the same connected component if and only if the component index `u->cc` is equal to the component index `v->cc`

```
sync void GraphTree(sh Graph *g, sh Tree *t);
```

`GraphTree` extracts a spanning forest of graph `g`; only valid after a call to `GraphCC`

```
sync void GraphGenedge(sh Graph *g);
```

`GraphGenedge` generates a random graph by constructing a set of `g->mm` randomly generated edges

```
sync void GraphGendeg(sh Graph *g, sh const int d);
```

`GraphGendeg` generates a random graph by constructing for `d` random edges for each node of `g`; edges may be discarded to restrict the total number of edges to `g->mm`

Table A.18 Dictionary Constructors and Destructors

Constants and Access Macros	
`DICT_A`	Value of a in (a, b) tree
`DICT_B`	Value of b; it must hold that $b = 2a - 1$
`DictSize(dict)`	Number of items in `dict`
`DictEmpty(dict)`	`dict` empty?
`DictItemContent(item)`	Content of dictionary item

The type of the key can be set at compile time to any type for which $<$ is defined; only the `typedef` definition of `keytype` must be changed

```
void NewDictItem(DictItem *item, keytype key, void *content);
```

`NewDictItem` initializes dictionary item `item`, assigning the item key `key` and content `content`

```
void DelDictItem(DictItem *item);
```

`DelDictItem` frees the contents associated with a dictionary item; space for content is assumed to have been allocated on the permanent shared heap

```
sync DictItem *DictBuild(sh DictItem *item[], sh int const n);
```

`DictBuild` builds an `n` item dictionary from the sequence of dictionary items in `item`; the items must be supplied in key order

```
sync void DictDestroy(sh DictItem *dictionary,
                      sh DictItem *item[]);
```

`DictDestroy` flattens and destroys `dictionary`, returning the ordered sequence of dictionary items (leaves) in array `item`

Table A.19 Dictionary Search, Insertion and Deletion Operations

```
DictItem *DictSearch(DictItem *dictionary, keytype key)
```

`DictSearch` searches `dictionary` for an item with key `key`; returns `NULL` if an item with key `key` is not present in `dictionary`

```
DictItem *DictClosest(DictItem *dictionary, keytype key)
```

`DictClosest` searches the dictionary for the item with the smallest key greater than or equal to `key`; if `key` is greater than the largest item in the dictionary, the largest key item is returned

```
DictItem *DictRetrieve(DictItem *dictionary, int k)
```

`DictRetrieve` returns a pointer to the `k`th smallest item of dictionary

```
sync DictItem *DictInsert(sh DictItem *dictionary,
                          sh DictItem *newitem[],
                          sh const int k);
```

`DictInsert` inserts a sequence of new items in key order into `dictionary`

```
sync DictItem *DictDelete(sh DictItem *dictionary,
                          sh DictItem *olditem[],
                          sh const int k);
```

`DictDelete` deletes a sequence of old items from `dictionary`; the items in `olditem` must appear in key order