SETL Newsletter Number 5 November 18, 1970
J. T. Schwartz

Miscellaneous algorithms written in SETL.

Here are various algorithms from Harrison's "Data Structures and programming" and from other sources written out in SETL (using the new external syntax).

1.) Linear lists.

A linear list is a set of items with a function next(item), such that next(item) = Λ for the last item.

Operations: insertion and deletion

```
define item insert, previtem; external next;
⟨next(item), next(previtem)⟩ ⟨next(previtem), item⟩; return;
end insert;
define delnext.item; external next; next(item) = next(next(item));
return; end delnext;
```

2.) Josephus problem (cf. Harrison notes; elementary algorithm)

Given integers $\{1 \leq \forall n \leq nn\}$, and initial m, produces sequence jos obtained by arranging in a circle, and successively removing every m-th.

```
jos = nl.; item = m-1; k = 1; next = {⟨n, if n lt. nn then n+1
else 1⟩, 1 ≤ ∀n ≤ nn};
(while next ne. nl.) jos(k) = next(item); delnext.item; (1 ≤ ∀j ≤ m)
item = next(item);;k = k+1; end while;
```

3.) Binary trees. A binary tree is a set of nodes with two descendant functions dright, dleft and a given top node ntop.

Operations: traversal. We take as an example the left-top-right

traversal order, and generate the sequence of nodes in order traversed.

```
seq = ∅; traverse.ntop; define traverse.top; external seq;
if top eq.Λ then return; traverse.dleft(top); seq(#seq+1) = top;
traverse.dright(top); return; end traverse;
```

Note that a very similar procedure would work for m-ary trees.

4.) Building a tree from a Polish string. Here a sequence of elements is to be built into a tree (which will have binary and triadic nodes). We assume that a function adic which tells us whether a given element of the Polish sequence is terminal, a monadic operator, or a binary operation is available; the function adic has the values 0, 1, 2 in these three cases.

```
definef polseq treebuild.adic; n = 1; m = 1; tree = nl.;
(while n lt.#polseq) item = polseq(n); a = adic(item);
if a eq. 0 then stack(n) = item; else node = newat.;
tree(node,1) = stack(m-p); tree(node,2) = item; if p eq. 2 then
tree(node,2) = stack(m-1);;
end else; n = n+1; m = m+1-p; return tree; end treebuild;
```

5.) Various sorting algorithms

Taken from recently distributed mimeographed notes on sorting (consult this document for algorithms)

5A.) Bubble sort. Rearranges a sequence seq in ascending order.

```
n = 1; (while n lt.#seq) if seq(n+1) ge. seq(n) then n = n+1;
```

else ⟨seq(n+1), seq(n)⟩ ⟨seq(n), seq(n+1)⟩ ; if n gt. 1 then
n = n-1;;
end else;;

5B.) Heapsort. Rearranges a sequence seq in ascending order by fast algorithm.

(2 ≤ ∀n ≤ #seq) m = n; (while m gt. 1 and. seq(m/2) gt. seq(m) )
⟨seq(m), seq(m/2)⟩ ⟨seq(m/2), seq(m)⟩; m = m/2;; end ∀n;
top = #seq; (while top gt. 1) til.out; ⟨seq(1), seq(top)⟩
⟨seq(top), seq(1)⟩ ;
m = 1; (while(2*m) lt. top) targ = if seq(2*m) lt. seq(2*m+1) and.
(2*m+1) lt. top then 2m+1 else 2m; if seq(m) lt. seq(targ) then
⟨seq(m), seq(targ)⟩ ⟨seq(targ), seq(m)⟩ ; m = targ; else go to out;;
end while; out:;

5C.) Ford-Johnson tournament sort.

This rather complex algorithm (for explanation, cf. sorting notes) minimizes the number of comparisons needed to sort a set of items. We assume a set items is given for sorting, on which a value-assigning function valf having numerical values is given. The routine produces a sequence consisting of the items arranged in increasing order of valf.

definef fordj.items; external valf;
if #items eq. 1 then return {⟨1,∋items⟩} else if # items eq. 2
then item 1 from.items; item 2 from.items;
if valf(item 1) lt. valf(item2) then ⟨item1,item2⟩ ⟨item2,item1⟩;;
return {⟨1,item1⟩, ⟨2,item2⟩}; else til fin;
map = nl.; items1 = nl.; (while items ne. nl.) item1 from.items;
item2 from.items;

```
if item2 eq.Λ then item2 = nl.;;
if valf(item1) lt. valf(item2) then ⟨item1,item2⟩⟨item2,item1⟩ ;;
item1 in.items1; map(item1) = item2; seq = fordj.items1; oseq = nl;
(1 ≤ ∀n ≤ #seq) oseq(n) = map(seq(n));; if oseq(#seq) eq. nl. then
oseq(#seq) = Λ ;;
oseq(1) insert.1; jbot = 2; jtop = 3; nelts = 3; (while oseq(jbot)
ne.Λ) jt = jtop;
(while oseq(jtop) eq.Λ) jtop = jtop-1;; oseq(jtop) insert.
(oseq(jtop) place.nelts);
jtop = jtop-1; if jtop lt. jbot then jbot = jt+1; nelts =
2*(nelts+1)-1;
jtop = nelts+1-jt;; end while; fin;;
define f eltplace.nelts; external seq,valf; bot = 1; top = nelts;
(while (top-bot) gt. 1) mid = top+bot/2; if valf(seq(mid)) gt.
valf(elt) then bot = mid; else top = mid;; end while; v = valf(elt);
return if v lt. valf(seq(bot)) then bot else if v lt. valf(seq(top))
then top else top+1; end place;
define elt insert.place; external seq;
seq = {⟨if n lt. place then n else n+1, seq(n)⟩} with. ⟨place,elt⟩ ;
return;
end insert; end fordj;
```

A set of algorithms for trees.

A threaded tree is represented by a set tree on which two functions r(node) and l(node) are defined, each for all but one node. One has r(node) = ⟨node',is⟩, where node' is either the

right descendant or the postorder successor, depending on whether is eq. t. or is eq. f. Similarly, l(node) = ⟨node',is⟩, where node' is either the left descendant or the postorder predecessor.

Traversal sequence. Traversing tree in postorder.

```
seq = nl.; node = top; dleft: (while - l(node)) node =* l(node);;
right: seq(#seq+1) = node; if r(node) eq. ∧ then go to done;;
dest = if-r(node}thendleft else right; node =* r(node); go to
dest; done:;
```

Insertion of elements in threaded tree. a.) Insert to right of node nod.

```
define elt putright.nod; ⟨r(elt),l(elt),r(nod)⟩ ⟨r(nod),⟨nod,f.⟩,
⟨elt,t.⟩⟩;return; end putright;
```

b.) Insert to left of node nod.

```
define elt putleft.nod; ⟨r(elt),l(elt),l(nod)⟩⟨⟨nod,f.⟩,l(nod),
⟨elt,t.⟩⟩; return; end putleft;
```

Delete subtree of threaded tree at given node.

```
define delnode.node; n = node; (while - l(n)) n =* l(n);;
ltarg =* l(n);
n = node; (while - r(n)) n =* r(n);; rtarg =* r(n);
if (*r(ltarg)) eq. node then r(ltarg) = ⟨rtarg,f.⟩;
else if (*l(rtarg)) eq. node then l(rtarg) = ⟨ltarg,f.⟩;
else l = nl.; r = nl.;;
end delnode;
```

Thread an unthreaded tree. First let seq be the postorder traversal sequence, as defined by previous algorithm. Then

suc = $\{\langle seq(n), seq(n+1)\rangle , 1 \leq n \leq \#seq-1\}$; pred = $\{\langle -x, *x\rangle, x \in suc\}$;

$(\forall n \in tree)$ l(n) = if l(n) ne.$\Lambda$ then $\langle l(n), t.\rangle$ else if pred(n) ne.$\Lambda$ then $\langle pred(n), f.\rangle$

else $\Lambda$; r(n) = if r(n) ne.$\Lambda$ then $\langle r(n), t.\rangle$ else if suc(n) ne.$\Lambda$ then $\langle suc(n), f.\rangle$ else $\Lambda$ ; end $\forall$ n;

Convert an m-ary ordered tree to a binary tree. First descendant becomes left descendant; next sibling becomes right descendant.

l = $\{\langle n, desc(n,1)\rangle , n \in tree / desc(n,1) \text{ ne.} \Lambda\}$;

r = $\{\langle desc(n,j), desc(n,j+1)\rangle , n \in tree, 1 \leq j \leq \#desc(n)-1\}$;

Convert binary tree to m-ary ordered tree. Inverse of preceding transformation.

desc = nl.;$(\forall n \in tree)$ j = 1; ll = l(n); m = n; (while ll ne.$\Lambda$);

desc(m, j) = ll; j = j+1; ll = r(ll); m = ll; end $\forall$ n;

Copy tree (note that this is general procedure for forming an isomorphic copy):

$(\forall n \in tree)$ copy(n) = newat.;; l = l u.$\{\langle copy(n), l(copy(n))\rangle, n \in tree\}$;

r = r u.$\{\langle copy(n), r(copy(n))\rangle , n \in tree\}$;

Enumeration of all binary trees with n nodes.

l = nl.; r = nl.; top = $\Lambda$; $(1 \leq \forall j \leq n)$ a = newat.;

$\langle l(a), top\rangle = \langle top, a\rangle$;;

(while top ne.$\Lambda$) print top; top = nextt.top;;

definef nextt.top; if l(top) ne.$\Lambda$ and.(nextt.l(top)) ne.$\Lambda$ then return top;

```
else if r(top) ne.Λ and.(nextt.r(top)) ne.Λ then l(top) =
newt. l(top);
return top; else if l(top) ne.Λ then ⟨save, l(top)⟩⟨l(top),
newt. l(l(top))⟩ ;
r(save) = r(top); r(top) = newt.save; else return Λ;;
define newt.top: t = top; (while r(t) ne.Λ '
⟨l(t), r(t), t⟩ = ⟨r(t), Λ, r(t)⟩ ;
return top; end newt; end nextt;
```